Parmys: Odin-II Intelligent Partial Mapper for Yosys Synthesis Suite

by

Daniel Khadivi

Bachelor of Computer Science–Software Engineering,
University of Tehran, 2017

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor:	Kenneth B. Kent, Ph.D., Computer Science
Examining Board:	Hung Cao, Ph.D., Computer Science, Chair
	Francis Palma, Ph.D., Computer Science
	Chris Rouse, Ph.D., Electrical and Computer Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

August, 2023

© Daniel Khadivi, 2023

Abstract

Partial mapping is to preallocate design logic at the beginning stages of the synthesis flow. Verilog-to-Routing (VTR) is an open-source FPGA architecture research framework. Odin-II provides Verilog elaboration and partial mapping for the VTR flow. Yosys is a Register-Transfer-Level (RTL) synthesis framework with extensive Verilog-2005 support. Partial mapping is possible in Yosys, but users are forced to manually determine the circuit implementation. Parmys automates the hard logic inference and the hard or soft mapping decisions for Yosys through the Odin-II intelligent partial mapper and brings architectural awareness to Yosys. This thesis discusses steps taken to add the Odin-II partial mapper into Yosys. Parmys expands research opportunities for Yosys users in the area of partial mapping considering FPGA architecture and device capacity. The standard VTR benchmarks showed improvement for all key quality-of-result (QoR) metrics when using the Parmys plug-in with the Yosys elaborator as the frontend in the VTR flow.

Acknowledgements

To my supervisor, Kenneth B. Kent. The completion of this research would not have been possible without your support and guidance. I am thankful for this wonderful opportunity to work on an industry-led project. I will never forget your kindness and help during my studies.

To my loving wife Mahsa. Without you, the completion of this work and the thesis document would have remained beyond reach. There is so much more waiting for us down the road, and I am grateful to have you by my side every step of the way.

To my dear parents. Without your support and perseverance, I would never have managed to get this far. From the bottom of my heart, thank you for always believing in me and pushing me to reach for my dreams.

To my brother, Aria, thank you for always being there with a sense of humour that can brighten any day.

This research was conducted within the Centre for Advanced Studies–Atlantic, Faculty of Computer Science, University of New Brunswick. The author is grateful for the colleagues and facilities of CAS–Atlantic in supporting this research. The author would also like to acknowledge the financial support of Google and the New Brunswick Innovation Foundation (NBIF).

I would like to express my gratitude to Stephen MacKay for his exceptional grammatical acumen and consistently valuable research recommendations.

Table of Contents

Abstract **ii**

Acknowledgments **iii**

Table of Contents **iv**

List of Tables **vii**

List of Figures **ix**

List of Algorithms **xi**

Abbreviations **xii**

1 Introduction **1**

1.1 Context 1

1.2 Problem Statement 3

1.3 Contributions 3

1.4 Outline 4

2 Background **5**

2.1 Y-Chart 5

2.2 Levels of Abstraction 5

2.3 FPGA Architecture 8

2.4 FPGA CAD Flow 9

2.5 Synthesis 11
2.6 Technology Mapping 11
2.7 Partial Mapping 12
2.8 VTR 13
2.9 Odin-II 13
2.9.1 Data Flow 15
2.9.2 Data Structures 15
2.10 Yosys 17
2.10.1 Data Flow 18
2.10.2 Data Structures 18
2.11 Yosys+Odin-II 20
2.12 Summary 21

3 Parmys Plug-in 22

3.1 Design 22
3.2 Algorithm 23
3.3 Read Architecture 25
3.4 Transform 27
3.5 Skip/Map 27
3.6 Partial Mapping Algorithm 29
3.7 Update 32
3.8 Optimization 33
3.9 FPGA Architecture File 34
3.10 Parmys Information Flow 35
3.11 Verification by Unit Tests 35
3.12 Passes 38

4 Parmys Frontend 41

4.1 Idea . . . 41
4.2 Conventional VTR Flow . . . 42
4.3 New VTR Flow . . . 42
4.4 Verification by Regression Tests . . . 42
4.5 New Regression Tests . . . 43
4.6 Odin-II Deprecation . . . 43

5 Results and Analysis **45**
5.1 Proof of Concept . . . 45
5.2 Parmys Frontend in the VTR flow . . . 46

6 Conclusions and Future Work **58**
6.1 Future Work . . . 59

Bibliography **67**

A VTR Benchmarks **68**

B Optimization in VTR Benchmarks **70**

C Koios Benchmarks **73**
C.1 Default . . . 73
C.2 No Custom Hard Blocks . . . 75
C.3 Multiple Architectures . . . 77

Vita

List of Tables

4.1 Benchmarks 44

5.1 VTR flagship architecture characteristics 47

5.2 Koios benchmarks using Yosys as the elaborator and the Parmys plug-in as the partial mapper in the VTR flow. 56

A.1 VTR benchmarks using Yosys as the elaborator and the Parmys plug-in as the partial mapper in the VTR flow. Results are relative to the baseline VTR setting. 69

B.1 VTR benchmarks using Yosys as the elaborator and the Parmys plug-in as the partial mapper in the VTR flow. Results are relative to the baseline VTR settings and the Geometric mean of the all circuits within each benchmark is reported. 71

C.1 VTR benchmarks using Yosys as the elaborator and the Parmys plug-in as the partial mapper in the VTR flow. Results are relative to the baseline VTR setting. 74

C.2 Koios benchmarks targeting `k6_frac_N10_frac_chain_depop50_mem32K_40nm.xml` architecture using Yosys as the elaborator and the Parmys plug-in as the partial mapper in the VTR flow. Results are relative to the baseline VTR setting. 76

C.3 The `conv_layer.v` circuit from the Koios benchmarks targeting 11 different architectures using Yosys as the elaborator and the Parmys plug-in as the partial mapper in the VTR flow. Results are relative to the baseline VTR setting. 78

List of Figures

1.1 Yosys simplified control flow [60] 2

1.2 Odin-II simplified control flow 3

2.1 Gajski-Kuhn Y-chart of design abstraction [32, 33, 42, 43] 6

2.2 Different levels of abstraction and synthesis. [60] 7

2.3 Generalized FPGA software flow [31] 10

2.4 Basic synthesis process [35, 60] 11

2.5 VTR CAD flow [45] 14

2.6 Odin-II simplified data flow (ellipses: data structures, rectangles: program modules) 16

2.7 Odin-II simplified entity-relationship diagram 16

2.8 Yosys simplified data flow (ellipses: data structures, rectangles: program modules) [60] 19

2.9 Yosys simplified entity-relationship diagram [60] 19

3.1 The Parmys pass integrated in the Yosys control flow. 23

3.2 Odin-II partial mapper integration as parmys Pass within Yosys data flow 24

3.3 Description of `add_2` circuit in Verilog format. 25

3.4 `add_2` circuit visualization in Yosys 25

3.5 `add_2` equivalent netlist in the Odin-II space after transform before partial mapping. 27

3.6 `add_2` netlist in the Odin-II space after partial mapping *(soft)*. 32

3.7 add_2 updated design in the Yosys space after partial mapping *(soft)*. Since no architecture is provided all adders are mapped to soft logic using primitive types. 34
3.8 add_2 netlist in the Odin-II space after partial mapping *(hard)*. 35
3.9 add_2 updated design in the Yosys space after partial mapping *(hard)*. 36
3.10 Definition of adder hard block in BLIF format. 37
3.11 Final flow 37
3.12 Chain of passes (green: Yosys standard passes, blue: Parmys passes) . 39

4.1 The new VTR CAD flow (Parmys, ABC, VPR) 43

5.1 Description of complex circuit in Verilog format. 46
5.2 complex circuit visualization in the Yosys space. 47
5.3 complex equivalent netlist in the Odin-II space after transform before partial mapping. 48
5.4 complex netlist in the Odin-II space after partial mapping. 49
5.5 complex updated design in the Yosys space. 50
5.6 Definition of mult_2_2_4 hard block in BLIF format. 51
5.7 Run-time comparison for the VTR flow utilizing the Odin-II synthesizer versus the Parmys synthesizer. 52
5.8 Memory comparison for the VTR flow utilizing the Odin-II synthesizer versus the Parmys synthesizer. 53
5.9 Number of hard-blocks comparison for the VTR flow utilizing the Odin-II synthesizer versus the Parmys synthesizer. 54
5.10 QoR comparison for the VTR flow utilizing the Odin-II synthesizer versus the Parmys synthesizer. 55

B.1 Run-time comparison for VTR benchmarks utilizing the Odin-II frontend and the Parmys frontend with various optimization settings. . . 72

List of Algorithms

3.1 Simplified Parmys algorithm 24
3.2 Read hard blocks from the architecture file and add definitions as module to the design 26
3.3 Transform Yosys design to Odin-II netlist 28
3.4 Decide whether to partial map or skip the node based on node type . . 29
3.5 Partial mapping algorithm 31
3.6 Update the design in the Yosys space 33

Abbreviations

AIG	And Inverter Graph
AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
BLIF	Berkeley Logic Interchange Format
BRAM	Block Random Access Memory
CAD	Computer Aided Design
CLB	Configuration Logic Block
DAG	Directed Acyclic Graph
DFF	D-type Flip Flops
DPRAM	Dual Port RAM
DSP	Digital Signal Processor
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
I/O	Input/Output
LUT	Look-Up Table
PARMYS	PARtial Mapper for Yosys Suite
QoR	Quality of Results
RAM	Random Access Memory
ROM	Read-Only Memory
RTL	Register Transfer Level
RTLIL	Register Transfer Level Intermediate Language
SPRAM	Single Port RAM
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit
VPR	Versatile Place and Route
VTR	Verilog to Routing
XML	Extensible Markup Language
YOSYS	Yosys Open SYnthesis Suite

Chapter 1

Introduction

Digital circuits are modelled at different levels of abstraction. The higher levels focus on the behavioural descriptions of the circuit *(architecture-agnostic)*, while the lower levels are primarily based on the implementation details *(architecture-dependent)*. Reducing the level of abstraction is called *synthesis*, and synthesis tools are usually applied to transform the Register Transfer Level (RTL) to an equivalent Structural Level (Gate Level) [35, 60]. An RTL design describes the data flow between high-level components, which are more architecture-agnostic. A gate level design describes the implementation using logical gates, which are more architecture-dependent.

1.1 Context

Partial mapping tries to bring some architectural awareness to RTL synthesis. It is defined as using available coarse-grained elements (like multipliers, adders, memories, or task-specific components) within the target device during synthesis and before technology mapping to increase the device utilization. Odin-II [39] is a Hardware Description Language (HDL) synthesis tool and provides intelligent partial mapping for hard blocks available within Field Programmable Gate Array (FPGA) architectures. The Odin-II partial mapper is capable of performing hard logic inference and

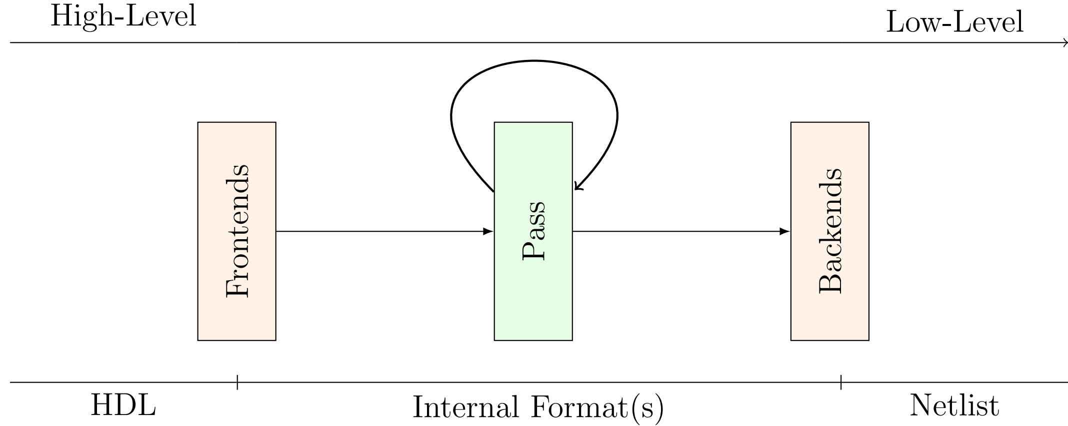

Figure 1.1: Yosys simplified control flow [60]

hard/soft trade-off [40] decisions (for multipliers) as to whether to map the logic to embedded hard blocks or soft logic gates.

Yosys Open SYnthesis Suite (Yosys) [59] is an open-source synthesis tool. Partial mapping is possible in Yosys, but users are forced to manually determine the circuit implementation. Yosys is unaware of the target architecture and users must make all the decisions themselves.

The fundamental difference between Yosys and Odin-II is that Yosys performs technology mapping in an iterative, incremental manner so that the user can map a subset of the design at each iteration (Figure 1.1), while Odin-II follows a recursive approach and completes the partial mapping in a single pass (Figure 1.2). More specifically, Yosys uses a hash map to store the references to all of the nodes in the circuit. On the other hand, Odin-II utilizes forward (child) and backward (parent) pointers to store the circuit tree and to reach a specific node the pointers from the root have to be traced in sequence. The Odin-II partial mapping algorithm uses a depth-first search (DFS) to traverse the whole circuit tree but the Yosys mapper can find any specific node within the circuit only by its name without traversing the whole circuit.

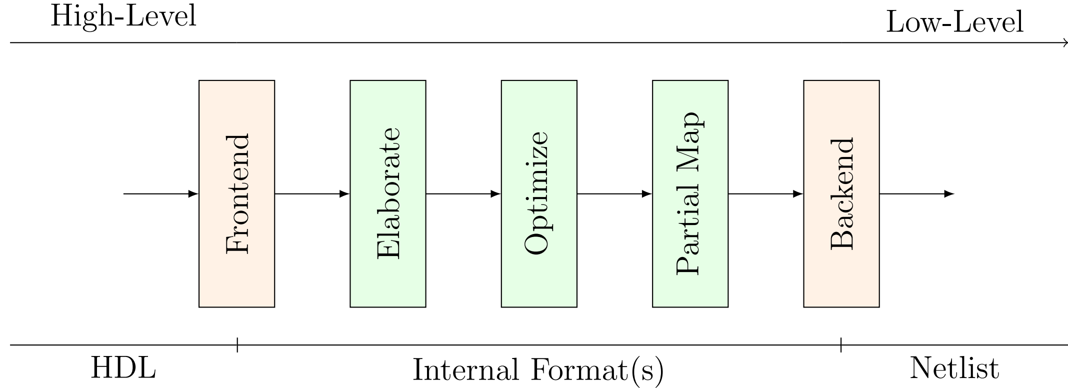

Figure 1.2: Odin-II simplified control flow

1.2 Problem Statement

Verilog to Routing (VTR) is considered the de-facto FPGA architecture research framework [55]. Odin-II, as the VTR frontend, is capable of automatic and intelligent coarse-grained mapping during synthesis. An architecture-aware partial mapper would improve device utilization by automating the decision of mapping discrete components into available hard blocks, using soft gates when no hard blocks are available, or mixing the soft/hard logic mapping for multipliers. Yosys supports coarse-grained mapping, but users must manually choose among circuit implementations.

1.3 Contributions

To add architectural awareness to Yosys, the necessary stages from the Odin-II partial mapper were extracted, refactored, and packed as a plugin for Yosys. The partial mapping features are accessible through the partial mapper for the Yosys suite (Parmys) plugin in Yosys, which parses the VTR FPGA architecture file and tries to map a set of predefined cell types (adders/subtractors, multipliers, memories, and Digital Signal Processing (DSP) blocks within the FPGA architecture) to hard-blocks or Yosys compatible soft-logic if no (or limited) hard-blocks are avail-

able. With Yosys becoming aware of the target FGPA's architecture, a systematic analysis of different target architectures is possible in Yosys.

Since Yosys and Odin-II have different approaches regarding netlist modelling (iterative vs. recursive), a skip procedure is added to the Odin-II partial mapper to skip certain types (simply any type other than adder, subtractor, multiplier, and memory) during the partial map. Also, since Odin-II has been designed to complete the partial mapping in a single pass, the final output has to change so as not to explode to look-up tables (LUTs) and remain at the RTL level to keep optimization opportunities on the Yosys side after the Parmys pass.

The Parmys plugin enabled the Odin-II intelligent partial mapping features (hard logic inference, logic binding, and hard or soft logic trade-offs) within the Yosys synthesis suite. First, the HDL is elaborated in Yosys and the FPGA architecture file is parsed by the Parmys plugin to identify available hard blocks in the target architecture and other necessary information, e.g., LUT width. Then, a transform stage is performed to convert the Yosys design to an equivalent Odin-II netlist. Next, Parmys tries to map a group of specific logical elements including custom hard blocks to the provided FPGA target. Finally, the mapped netlist is used to update the Yosys design. Further operations may then be performed within Yosys or the Parmys pass can be called again if needed.

1.4 Outline

This thesis has five main chapters. Chapter 2 covers the background material required to understand the problem and the solution. Chapter 3 elaborates the design and implementation of the Parmys plugin. Chapter 4 introduces the Parmys frontend in the VTR flow. Discussion of the results can be found in Chapter 5. Chapter 6 presents the conclusions and possible future directions for research and development.

Chapter 2

Background

The following concepts are required to understand the motivation behind this thesis and the solution to the problem.

2.1 Y-Chart

To formalize large digital systems design and to represent specification domains, abstraction levels, and the relationship among them, Gajski-Kuhn proposed the Y-Chart [32, 33, 42]. Hierarchical levels of the design process are represented within five concentric circles (Figure 2.1). The abstraction level decreases from the outer to inner circle. Temporal and functional behaviour of the system are described in the *Behavioural Domain* without any reference to the implementation. The *Structural Domain* shows necessary building blocks of the system and their interconnections. Implementation details are described within the *Physical Domain* [34, 42].

2.2 Levels of Abstraction

To better address abstraction levels in Hardware Description Language (HDL) synthesis, the *Behavioural Level* is considered between the *Algorithm* and the *Register*

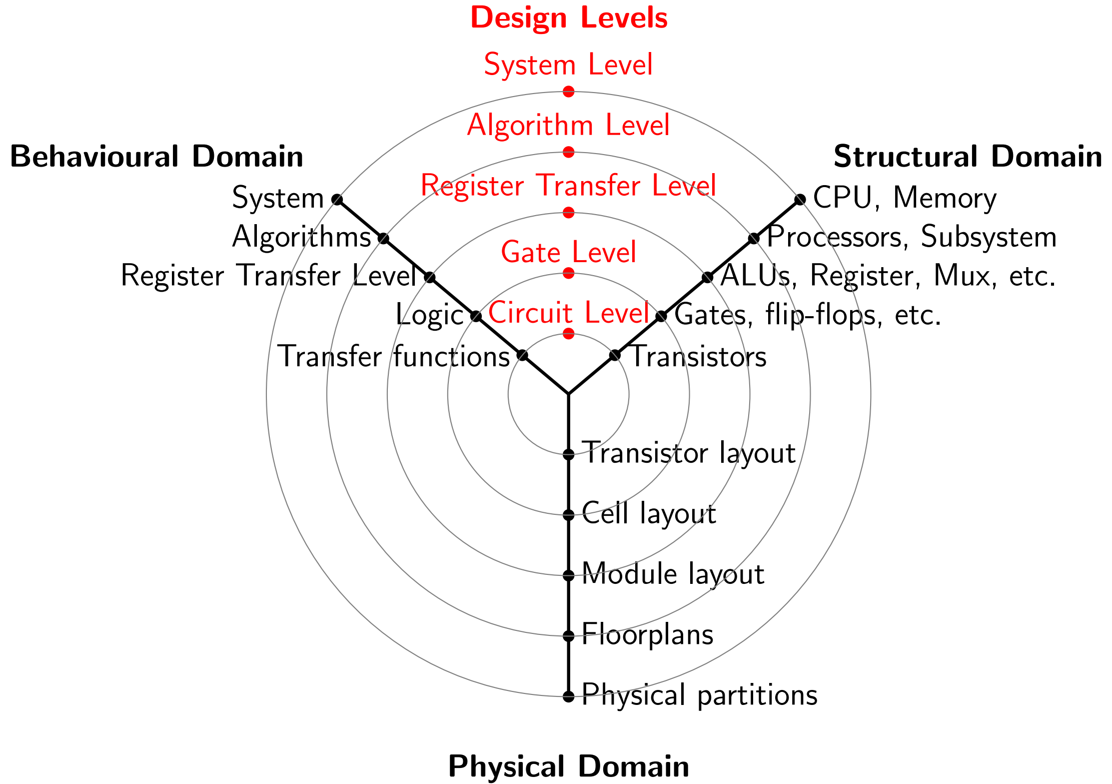

Figure 2.1: Gajski-Kuhn Y-chart of design abstraction [32, 33, 42, 43]

Transfer levels. Moreover, to enable both architecture-independent and architecture-dependent optimization and technology mapping, the *Gate* level usually is divided to *Logical Gate* and *Physical Gate* levels (Figure 2.2).

Design levels are categorized in five different levels:

- System Level: The system level is the most abstract level of design and describes the overall system structure and information flow. The system level specifications are converted to the algorithm level through system design or system-level synthesis [19].

- Algorithmic Level: The algorithmic level abstraction of a system (also referred to as *high* level or *architecture* level) defines a mapping between a set of inputs and outputs [19]. At this level the system is described by high-level programming languages. SPARK [36], GAUT [28], ROCCC [56], and LegUp [27] are

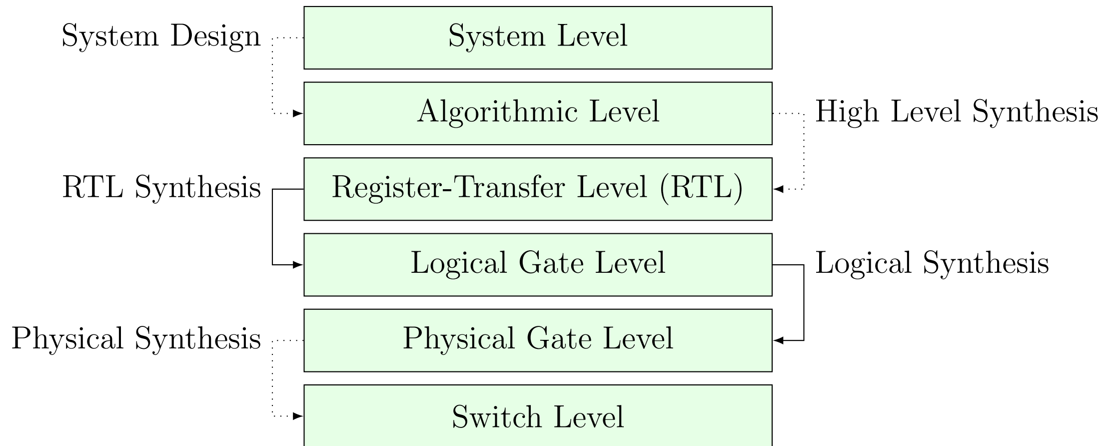

Figure 2.2: Different levels of abstraction and synthesis. [60]

academic high-level synthesis tools based on the C programming language [19].

- Register Transfer Level (RTL): At the RTL level the circuit is described using a graph of registers, combinatorial nodes, and signals [43]. Two common HDLs to model the design at the RTL level are Verilog and Very High-Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL) [18, 60].

- Gate Level: At the gate level, the behaviour of RTL components is defined by gates and registers. To better discriminate the architecture-independent and architecture-dependent operations like optimization and technology mapping, the circuit at the gate level is described by a logical gate level design and a physical gate level design.

  - Logical Gate Level: The logical gate level of abstraction describes the circuit by a netlist of single-bit basic logical gates and registers. *Two-level logic* synthesis is the simplest approach in which the boolean logic is converted to a sum-of-products form and later is represented by a maximum of two logic gates between input and output. Another approach is *multi-level logic* synthesis where two common algorithms are And-Inverter-Graph (AIG) [41, 51] and Binary-Decision-Diagram (BDD) [26].

  AIG uses directed acyclic graph (DAG) of two input ANDs and inverters [48]. ABC [25], MVSIS [11], and AIGER [22, 23] are open-source tools for multi-level logic synthesis. BDD is based on a binary tree of decision nodes.

  - Physical Gate Level: The physical gate level is represented using a netlist of physically available gates and registers within the target architecture. For a Field Programmable Gate Array (FPGA), the physical gate level is look-up tables (LUTs) with optional output registers [60].

- Circuit Level: At the circuit level, also known as the switch level or the transistor level, the gate level is implemented by a netlist of single transistors [19].

2.3 FPGA Architecture

FPGAs provide software flexibility alongside hardware performance. In other words, FPGAs are programmable hardware, and due to their potential, they have found their way into embedded systems. The fundamental elements of FPGAs are:

- Configurable Logic Blocks (CLBs): perform logical operations and provide storage.
- Input/Output (I/O) blocks: import and export data through physical ports.
- Wires: provide connectivity among all other elements.

Blocks and signals are routed through a network of programmable interconnects named a fabric. CLBs are based on one or more LUTs. FPGAs are expected to provide both logic and storage through a set of basic components referred to as configurable logic blocks (CLBs). The logic can be either fine-grained (a transistor) or coarse-grained (an entire processor). To reach a trade-off, LUT-based CLBs are

being used by commercial vendors (like Xilinx [15]—acquired by AMD [3]—and Altera—acquired by Intel [8]) and include `NAND` gates, multiplexers, and LUTs [31]. In modern heterogeneous FPGAs, coarse-grained elements like multipliers, adders, memories, and processors can be found, which are referred to as black boxes or Hard Blocks (HBs) [64]. HBs serve specific fixed functionalities [57]. The number and size of the elements and their connections are called an FPGA architecture. Complex embedded blocks may be found within domain-specific FPGA architectures [64], and will be referred to here as Digital Signal Processing (DSP) blocks.

FPGAs provide common storage elements like Random-Access Memory (RAM), Read-Only Memory (ROM) and shift registers through specific physical elements, named block RAMs (BRAMs), LUTs, and shift registers. BRAM is a dual port RAM to provide parallel memory access. ROM is implemented using a block of RAM. Shift registers allow reuse of data along a computational path [19].

2.4 FPGA CAD Flow

A Computer Aided Design (CAD) flow converts a Hardware Description Language (HDL) into a bit stream that can be loaded into an FPGA (Figure 2.3). To maximize the hardware utilization in a feature-rich FPGA architecture, it is crucial for the CAD tool to take advantage of the hard blocks [31].

Packing (clustering) is the process of grouping k-input CLBs to clusters that are directly mappable into logical elements on an FPGA. The *Placement* algorithm decides which logic block within the FPGA should be used for each cluster. Allocating routing resources to nets is performed at the *Routing* stage [31].

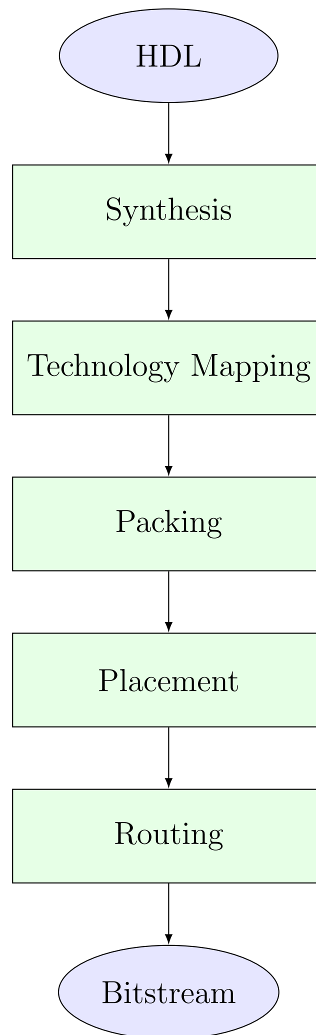

Figure 2.3: Generalized FPGA software flow [31]

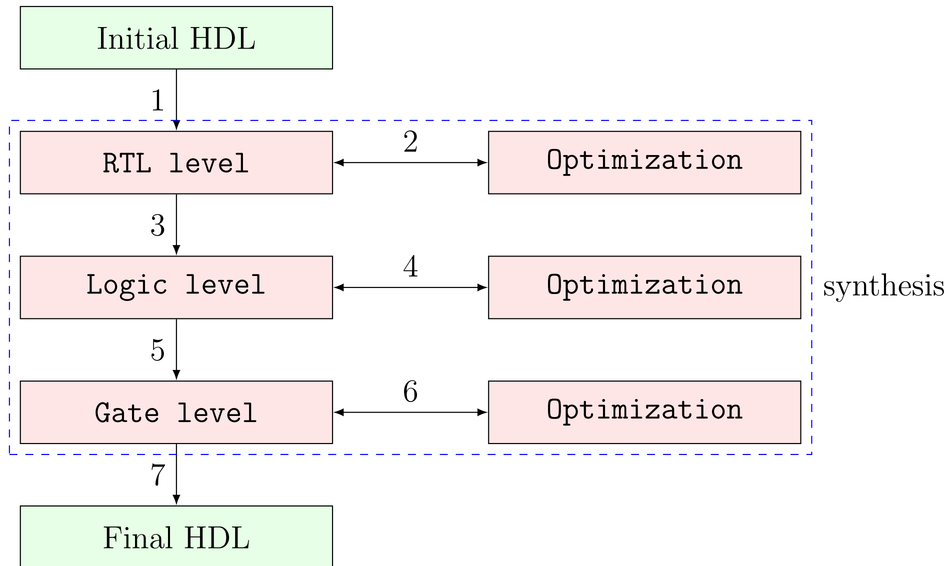

Figure 2.4: Basic synthesis process [35, 60]

2.5 Synthesis

Reducing the abstraction level of the design is called *synthesis* (Figure 2.4). At *logic synthesis* an HDL (VHDL or Verilog) is converted to a network of primitive gates and flip-flops [24]. First HDL is translated to an RTL-level netlist, then the netlist is transformed to an equivalent netlist of logical gates and finally the logical gates are transformed to physically available gates within the target device. At each stage of the synthesis process, optimization is used to increase the performance of the design. Verification steps are also useful to verify the model at each level of abstraction [35].

2.6 Technology Mapping

The synthesis process first transforms the HDL code to an RTL netlist [63], which is based on abstract coarse-grained cell types, and the technology mapping refers to the process of converting an RTL netlist to a netlist of available cell types in the target hardware [60].

The critical path of a circuit is the set of logic gates, interconnects, and registers

that have the highest cumulative delay. The delay of the critical path determines the maximum operating frequency or the minimum clock period of the circuit. Reaching optimal area usage and minimum critical path delay are two goal functions of a technology mapper by choosing the most area-efficient and the fastest set of gates to map the logic [17].

One big challenge is to decide which coarse-grained cell to use to express a function in coarse-grained synthesis as there are many ways to perform it [61]. Technology mapping usually takes place in two steps. First, RTL cells are converted to equivalent single-bit cells and later mapped to available gates in the target device [60].

2.7 Partial Mapping

If a specific functionality is being used repeatedly, then using a hard implementation in favour of a soft one may increase the application's performance [47]. For example, if a circuit has numerous addition operations, then targeting an FPGA device that contains hard block addition operators would prove advantageous. After the first stage of technology mapping, all the coarse-grained information is lost, and if coarse-grained blocks are available within the target FPGA architecture, the RTL netlist has to be mapped to them directly before any further actions in technology mapping [38, 60].

Usually, the best stage in which to perform coarse-grained mapping is one of the early stages of a CAD flow, like the RTL synthesis level where coarse-grained structures are easier to recognize directly, and before later stages like technology mapping [38]. Awareness of the FPGA architecture is necessary during partial mapping to maximize device utilization. In other words, partial mapping is performing some technology mapping, but in advance. The partial mapper binds some parts of the netlist to the target technology while the rest of the netlist remains intact to be mapped by

logic synthesis [38].

2.8 VTR

Verilog-to-Routing (VTR) [45, 49], is an open-source Computer-Aided Design (CAD) tool and uses the human-readable Extensible Markup Language (XML) standard to describe an FPGA architecture [62]. In a basic scenario (Figure 2.5), Verilog and architecture files are inputs into the flow, Odin-II as the frontend reads and elaborates the design. ABC [2, 25] performs the synthesis and technology mapping, and later VPR [21, 46] is in charge of packing, placement, routing, and analysis.

In addition to elaboration, Odin-II tries to preallocate the design to available hard blocks in the FPGA architecture [38].

2.9 Odin-II

Odin-II serves as both a Verilog frontend and partial mapper for VTR. Odin-II supports hard blocks like block-RAM, carry-chains, and multipliers but can only export the partially mapped netlist to a BLIF (Berkeley Logic Interchange Format) file, which is a textual representation of the logic-level hierarchical circuit [5].

Being architecture-aware provides the opportunity for systematic analysis of possible innovations in theoretical FPGA architectures [37]. Odin-II also comes with a simulator, which is widely used for the evaluation of benchmarks [52, 54] and functional verification [44, 50]. Odin-II is known for *hard/logic inference*, which means reading and parsing the FPGA architecture file and taking advantage of the available coarse-grained blocks within the target device during partial mapping.

Moreover, Odin-II performs intelligent partial mapping, which is defined as making decisions regarding the *hard/soft trade-offs*. To be more specific, if a limited number of hard blocks are available within the architecture, the rest of the logic will be

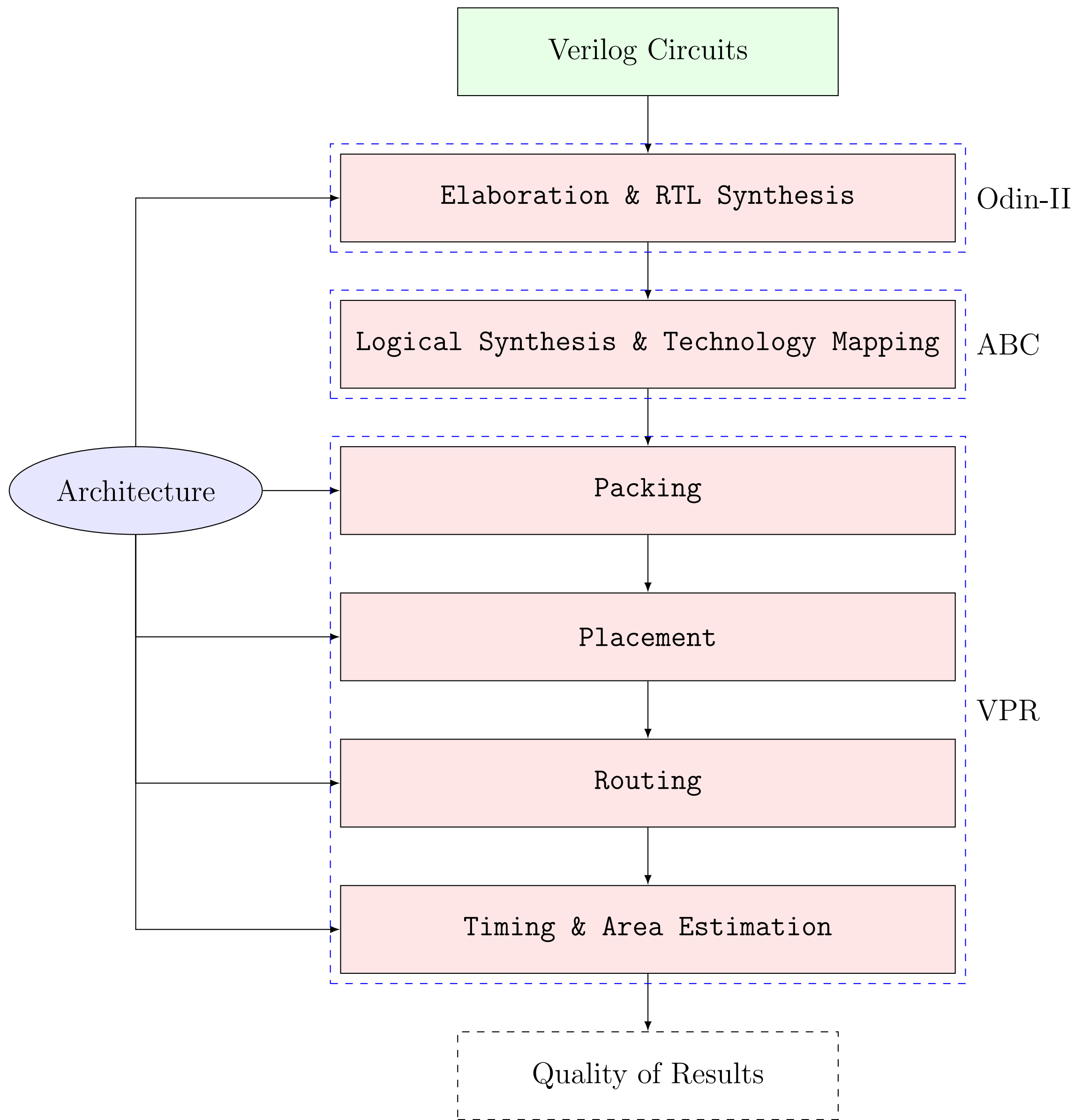

Figure 2.5: VTR CAD flow [45]

mapped to fine-grained blocks (soft logic) [40]. Although Odin-II provides intelligent partial mapping, it does not fully support all of the synthesisable Verilog [37, 55].

2.9.1 Data Flow

Figure 2.6 demonstrates the simplified data flow of Odin-II. First the HDL design is parsed by the Verilog frontend and an abstract syntax tree (AST) is created, then the AST is elaborated and an equivalent netlist is created. The netlist elements are either generic gates or generic black boxes like memories, multipliers, adders, or subtractors [57]. After optimization, the netlist and the FPGA architecture are passed to the partial mapper. The partial mapper first infers available coarse-grained elements within the FPGA architecture and then preallocates the logic upon availability. At the end, the BLIF backend outputs a flattened netlist [39]. When instances of all modules within the netlist, other than the top module, are replaced with their actual implementations recursively, the netlist is called flattened.

2.9.2 Data Structures

The main data structures of Odin-II and the relationship among them are visualized in Figure 2.7. In Odin-II, the circuit should be flattened, in other words, it can only have a single module that is represented as the *netlist* structure. It can be seen as a graph of the top input nodes, top output nodes, and all other internal nodes that are connected together via *nets* and *pins*.

To represent the connections among nodes, the wires are modelled as a *pin* structure. A pin can be of the input or output type. Input pins exit a node and enter a net, while output pins exit a net and enter a node. Virtually, a set of pins can be seen as a *port*, but Odin-II does not have any specific data structure for ports.

A `node` is the representation of the logical cell of a certain type, and each node owns a few input and output pins. Each input pin entering a node comes out of a *net*,

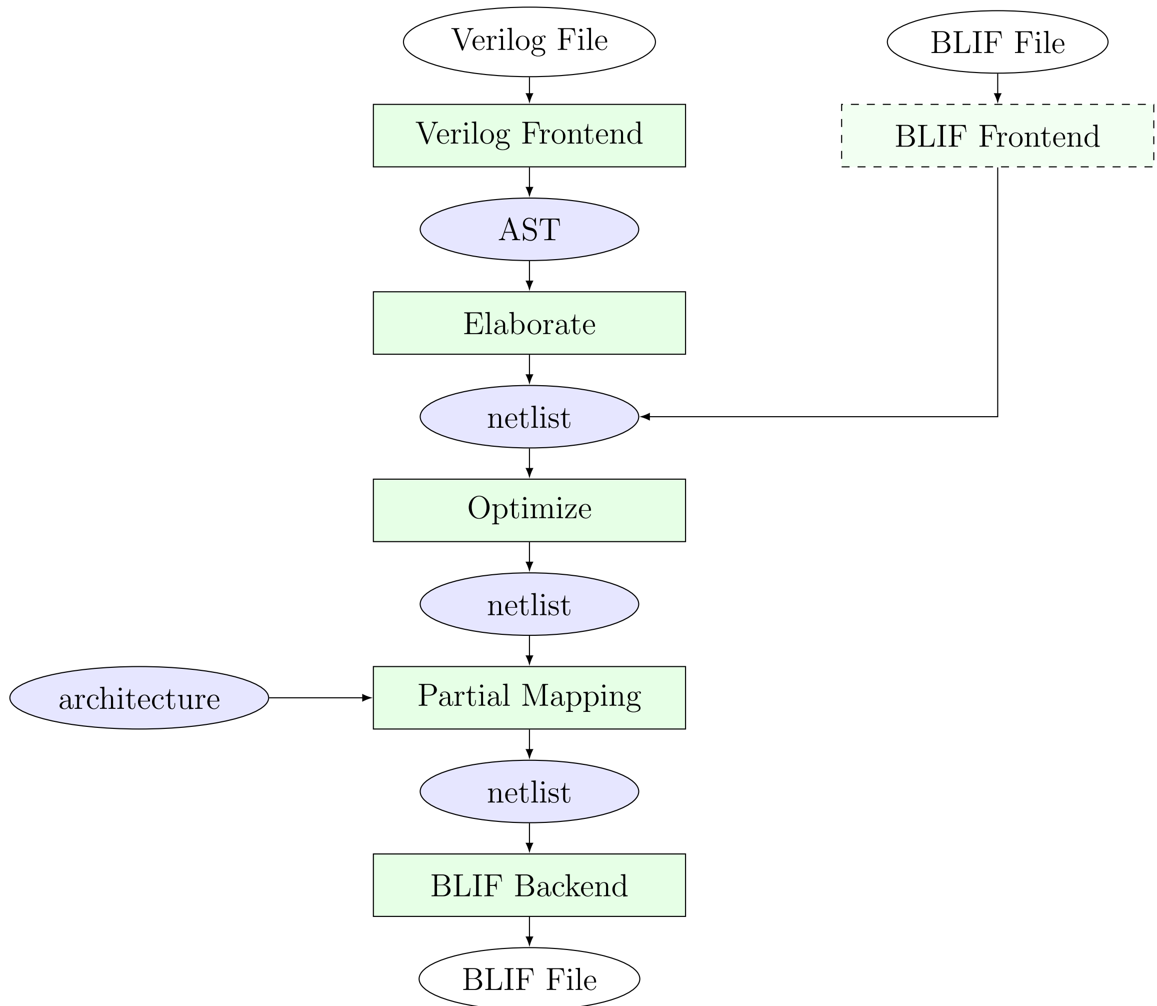

Figure 2.6: Odin-II simplified data flow (ellipses: data structures, rectangles: program modules)

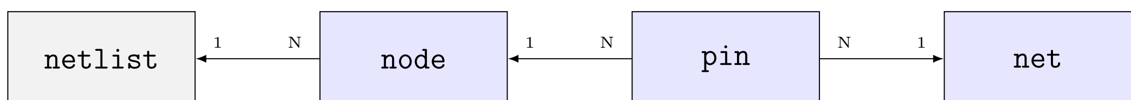

Figure 2.7: Odin-II simplified entity-relationship diagram

called a `fan out pin`, and each output pin going out of a node and entering another net is called a `driver pin`. In Odin-II pins are connected to each other indirectly through nets [52].

2.10 Yosys

Yosys (Yosys Open SYnthesis Suite) [58] is an open-source RTL synthesis framework with various frontends and backends. Algorithms are implemented as passes and synthesis jobs are possible by combining the existing passes. Yosys converts the description of behavioural design to RTL, logical gates, or physical gates [59]. Yosys uses ABC to perform technology mapping. Yosys can partially map, but the user is forced to choose among different circuit implementations manually. Also, technology mapping has no automated procedures to mix hard/soft logic.

Yosys consists of three types of components: frontend, pass, and backend. A frontend reads a specific input format and converts it to an RTL-level netlist called the Register-Transfer-Level-Intermediate-Language (RTLIL), which is the internal format used by Yosys to model digital circuits. The RTLIL representation is divided into two subcategories: RTL and Gate. The Yosys RTL cells are used for coarse-grain logic like adders with multi-bit input and output signals. The Yosys Gate cells are utilized for fine-grain logic like primitives with single-bit inputs and outputs. RTLIL is not a standard format and contains Yosys-specific types which are not known to external tools.

Any RTL-level algorithm can be implemented as a pass in Yosys to update the RTLIL design. Each pass performs a specific functionality on a subset of cells. Yosys has numerous predefined passes, for example the optimization pass, which calls other type specific optimization passes to perform a complete optimization. The user may pass parameters to various passes to control performance of the operation.

A backend outputs the design to a specific format. Yosys supports Verilog, BLIF, and Electronic Design Interchange Format (EDIF) (which is a vendor-neutral format to express digital design) backends to output the netlist [60]. Producing BLIF output from Yosys Gate cells is straightforward, but Yosys RTL cells are not available within the standard BLIF, hence, these cells are defined as black boxes in the output, which results in an unknown implementation.

2.10.1 Data Flow

Figure 2.8 demonstrates the simplified data flow of Yosys. First the Verilog frontend parses the Verilog input and creates an AST, then the AST frontend consumes the AST and creates an equivalent RTL level netlist. The RTLIL representation supports registers, logical gates, basic behavioural constructs, and multi-port memories. Any RTL-level algorithm can be implemented as a Pass in Yosys to update the RTLIL Design.

2.10.2 Data Structures

Yosys is implemented in C++. In Yosys, *Design* is a container for modules. It also holds references to connections among different module instances within the circuit. The *Module* construct is used to represent Verilog modules in Yosys. A module may be marked as a black box [60].

Cells represent logical units of various types. Each cell has a set of attributes that limits or extends its functionalities. Yosys cells are either *internal RTL cells* or *internal gate cells*, the first one is used for coarse-grained logic (including signal vectors and complex cells like adders) and the second one is used for fine-grained logic (including single bit signals and primitives) [60] (Figure 2.9).

Alongside Cells, the *Wires* are used to model the underlying netlist. A Wire represents a signal of width 1 or more. A wire can also be a port of a module, in this case,

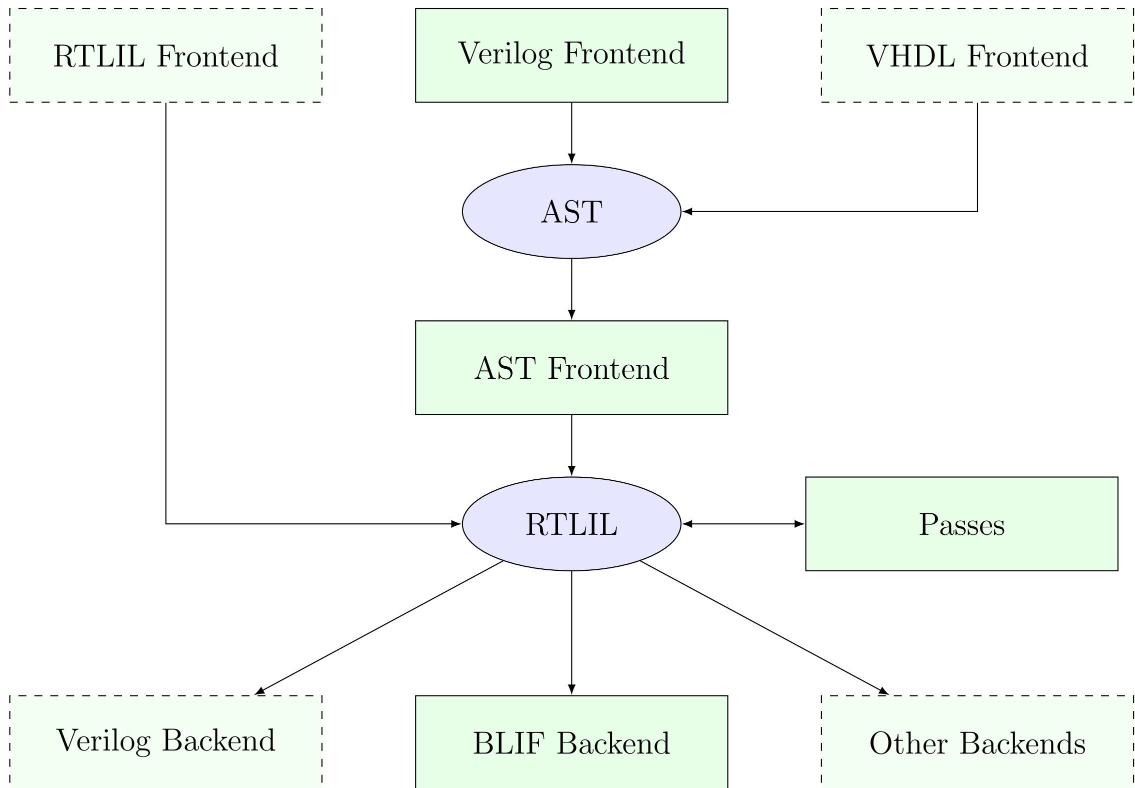

Figure 2.8: Yosys simplified data flow (ellipses: data structures, rectangles: program modules) [60]

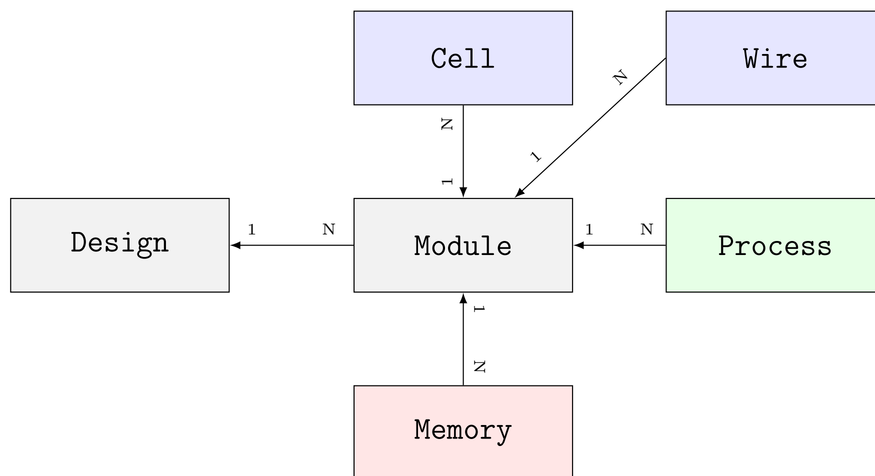

Figure 2.9: Yosys simplified entity-relationship diagram [60]

it should be input, output, or inout. Wires are used to connect different cells within a module. The *SigSpec* is a data structure to model signals. The underlying signal can have one or more wires. Also, the SigSig type consists of a pair of SigSpec objects to model the connections between wires which are represented with SigSpec [60].

The *Process* objects are inferred from high level behavioural constructs like *switch* and *always* identifiers within the HDL. Process constructs have to be converted to RTL-level elements to be synthesized or mapped. The *proc* pass performs behavioural synthesis on the Process objects [60].

The *Memory* objects are used to model storage at the behavioural level. After behavioural synthesis a Memory construct is converted to a cell type of either *$memrd* (for read accesses) or *$memwr* (for write accesses). These cells can later be converted to D-type Flip-Flops (DFFs) or multi-port memory blocks named *$mem* by the *memory* pass. The RTL-level memories in Yosys are candidates to be mapped into hard memory blocks within the target architecture [60].

2.11 Yosys+Odin-II

Yosys+Odin-II [29] was implemented to extend Verilog support in the Odin-II Verilog frontend, since Odin-II lacks support for the Verilog-2005 standard. First Yosys reads the input Verilog descriptor file and generates an intermediate BLIF file, which consists of coarse-grain RTLIL Yosys internal types. Originally Odin-II could not read the Yosys output and could only be parsed with the Yosys BLIF frontend. To overcome this issue, the BLIF frontend in Odin-II was modified to read coarse-grain Yosys compatible types (e.g., $add) from the intermediate BLIF input file and convert them to Odin-II compatible types (e.g., adder). After BLIF elaboration, Odin-II continues to perform partial mapping within the VTR flow as usual. As there are fundamental differences between Yosys types and Odin-II types, many

translation steps were added to Odin-II to make it compatible with a subset of Yosys types that are likely to show up in the intermediate BLIF file. The downside is that if a new type is added to Yosys, Yosys+Odin-II could break since not all of Yosys types were added to Odin-II. Moreover, restrictions were applied when Yosys reads the Verilog input to prevent the presence of certain logic types in the intermediate BLIF file which are not translatable within Odin-II, accounting for missing optimization opportunities within Yosys.

2.12 Summary

In this chapter, Y-Chart and digital design abstraction were described. FPGA, FPGA architecture, and FPGA CAD flow were discussed. Various steps of synthesis, technology mapping, along with partial mapping were reviewed. The different tools and components that will be used in integrating the Odin-II intelligent partial mapper within the Yosys synthesis suite were also presented. In the next chapter the proposed design along with implementation details will be presented.

Chapter 3

Parmys Plug-in

Parmys is designed to bring VTR-style architectural awareness to Yosys Open SYnthesis Suite (Yosys). With Parmys being plugged-in to Yosys, users can input the target Field Programmable Gate Array (FPGA) architecture file along with the circuit description. Parmys infers available hard blocks within the target FPGA and adds their signature to Yosys during the synthesis flow. So Yosys will understand if an element is defined of a previously unknown type, since the type is known to Yosys with help from Parmys. Later in the synthesis flow, when the user triggers the partial mapping to be performed, Parmys automatically takes care of mapping the arithmetic, memory, and custom Digital Signal Processing (DSP) cells to available hard blocks within the target device. With Parmys, Yosys becomes aware of the target device architecture, and the final design will better utilize the available hard blocks to improve the performance of the circuit implementation.

3.1 Design

The Parmys plugin automates the process of identifying coarse-grained elements and choosing between hard or soft implementations. The Parmys pass starts with transform, which is the initial step that converts the Yosys design to an equivalent

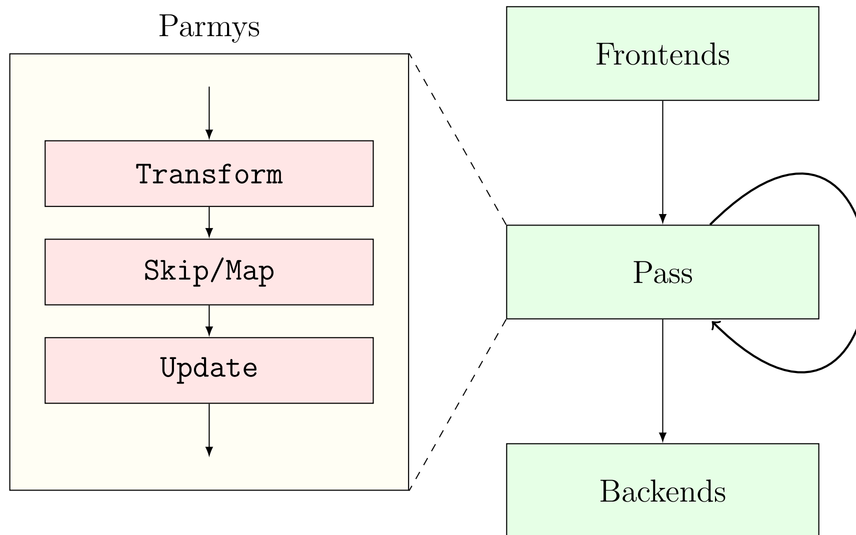

Figure 3.1: The Parmys pass integrated in the Yosys control flow.

Odin-II netlist. The partial mapper then decides whether to map the specific logic types to hard or soft blocks or skip them. Finally the mapped netlist will be used to update the Yosys design (Figures 3.1 and 3.2).

3.2 Algorithm

The Parmys algorithm starts with reading and parsing the FPGA architecture file. After retrieving necessary information from the target device, Yosys is ready to read and optimize the input circuit to create an Register Transfer Level (RTL) design. The design is transformed to an equivalent Odin-II netlist that is fed into the modified partial mapping algorithm to map a subset of nodes. The mapped netlist is used to update the design to reflect the output of the partial mapping stage. The process can be followed with other operations including, but not limited to, optimization (Algorithm 3.1).

For better understanding a simple circuit of a 2-bit adder is chosen to show basic functionality. The corresponding Verilog file and visualization are given in Figures 3.3 and 3.4 respectively. Later, a more complex example including Yosys internal

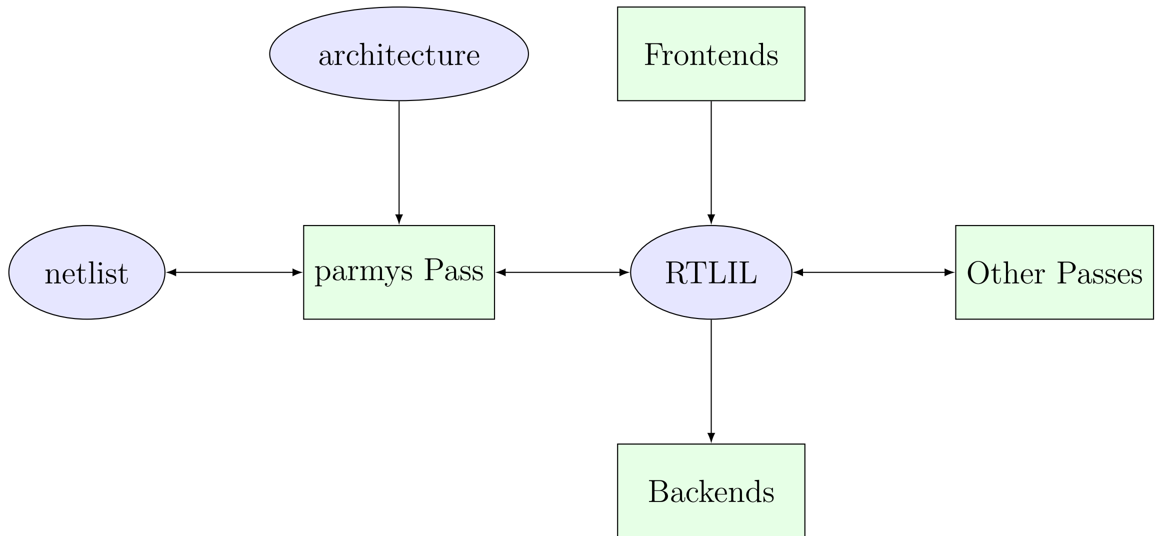

Figure 3.2: Odin-II partial mapper integration as parmys Pass within Yosys data flow

---

Algorithm 3.1: Simplified Parmys algorithm

Data: input circuit
Data: FPGA architecture file
Result: partially mapped design
1. *hard blocks* ← *read FPGA architecture*;
2. *design* ← *read input circuit*;
3. optimize the *design*;
4. *netlist* ← *transform*(*design*);
5. partial map the *netlist*;
6. *design* ← *update*(*netlist*);
7. optimize the *design*;

```verilog
`define WIDTH 2

module add_2(A, B, O);

input [`WIDTH-1:0] A,B;
output [`WIDTH-1:0] O;

assign O = A + B;

endmodule
```

Figure 3.3: Description of `add_2` circuit in Verilog format.

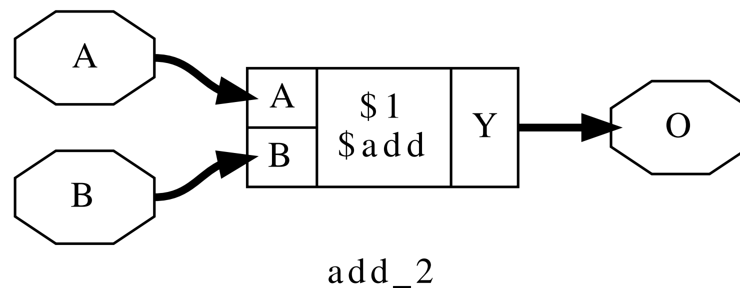

Figure 3.4: `add_2` circuit visualization in Yosys

cells and black box modules will be discussed. Odin-II follows a recursive approach along with doubly linked lists to model the netlist, whereas Yosys has adopted an iterative approach using a map data structure to model the underlying netlist. Given fundamental differences in netlist modelling and the fact that wires in Yosys do not hold back-references to cells, the best approach to integrate the Odin-II partial mapper with Yosys is to use a transform.

3.3 Read Architecture

Parmys utilizes the `libarchfpga`[1] library from VTR to read and parse the FPGA architecture file provided by the user. The `libarchfpga` library retrieves all necessary information from the architecture description file and stores it within specific data

[1] https://github.com/verilog-to-routing/vtr-verilog-to-routing/tree/master/libs/libarchfpga

structures, which are accessible through the synthesis process. Parmys receives the architecture information and iterates over all of the available hard blocks within the target architecture and adds each one as a new module type to Yosys (Algorithm 3.2).

Algorithm 3.2: Read hard blocks from the architecture file and add definitions as module to the design

Data: FPGA architecture file
Data: Design
Result: Hard blocks added as module to Yosys design

```
1  models ← hard blocks defined in target;
2  foreach hb ∈ models do
3      module ← new Module;
4      module.name ← hb.name;
5      foreach input ∈ hb.inputs do
6          port ← new Port;
7          port.input ← true;
8          port.wires ← input.wires;
9          add port to module ports;
10     foreach output ∈ hb.outputs do
11         port ← new Port;
12         port.output ← true;
13         port.wires ← output.wires;
14         add port to module ports;
15     module.black_box ← true;
16     add module to design;
```

Hard blocks are custom defined logic and before instantiating any cell of an unknown type, the type has to be declared within Yosys. Adding the signature of each available hard block to Yosys provides information about names and numbers of inputs and outputs and the type of the block. When a custom type is added to Yosys as a black box, Yosys only cares about the inputs and outputs and not the unknown logic within the box. So when a cell is defined to be of a custom type which has been introduced to Yosys, it is instantiated without issue and the inputs as well as the outputs are connected to other cells.

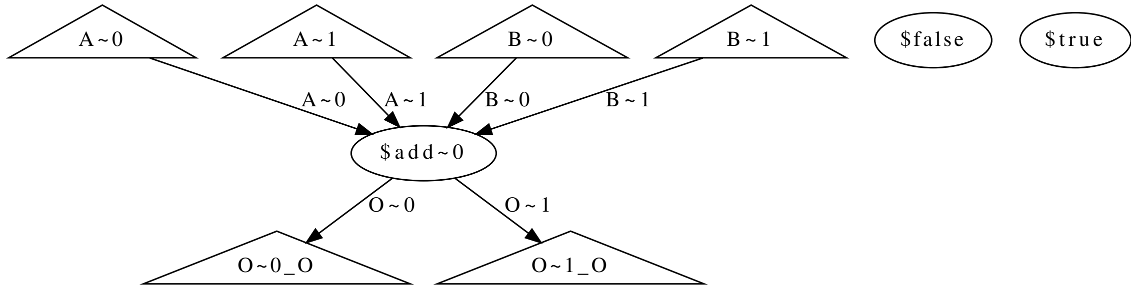

Figure 3.5: `add_2` equivalent netlist in the Odin-II space after transform before partial mapping.

3.4 Transform

After parsing the FPGA architecture XML file, all DSP blocks except for the VTR primitives (adder, multiply, single_port_ram, and dual_port_ram) are converted to black box modules and added to the design space. If necessary, the VTR primitives can also be added as whitebox modules to the design. At this stage the design from Yosys is traversed and converted to an equivalent netlist in the Odin-II space. Input wires, output wires, and cells from the top module are transformed to top input nodes, top output nodes, and internal nodes in the netlist (Algorithm 3.3).

Figure 3.5 demonstrates the transformed equivalent netlist of the sample input design. Each of the Yosys internal types (except for $add, $mult, $sub, and $mem) and any previously mapped cells are marked to be skipped during the partial mapping phase. After creating an equivalent Odin-II compatible netlist from the Yosys design, the transformed netlist can be handed forward to the next stages.

3.5 Skip/Map

As mentioned in Section 3.4, only certain types will be mapped and others will be skipped to reach maximum efficiency and preserve the optimization opportunity for Yosys because the whole synthesis flow is being performed within the Yosys ecosystem and Yosys better knows how to optimize the whole design. At this stage,

Algorithm 3.3: Transform Yosys design to Odin-II netlist

```
   Data: design
   Data: netlist
   Result: An equivalent netlist of nodes in Odin-II space
 1 netlist ← allocate_netlist();
 2 add VCC, GND, and PAD to netlist as top input drivers;
 3 top_module ← design.top_module;
 4 foreach wire ∈ top_module.wires do
 5     if wire is input then
 6         create input node from wire;
 7         add input node to netlist;
 8     else if wire is output then
 9         create output node from wire;
10         add output node to netlist;
11 foreach cell ∈ top_module.cells do
12     node ← allocate_node();
13     node.params ← cell.params;
14     node.type ← infer_type(cell.type);
15     foreach conn ∈ cell.connections do
16         if conn is input then
17             create input node from conn.wire;
18             add input node to netlist;
19         else if conn is output then
20             create output node from conn.wire;
21             add output node to netlist;
22     add node to netlist as an internal node;
23 foreach node ∈ netlist do
24     foreach input ∈ node do
25         source ← find_source(input);
26         connect input to source node;
```

the Odin-II partial mapper with respect to the device capacity will try to maximize the hardware utilization through mapping certain types to available hard blocks and if no hard blocks are available, the logic will be exploded to gates (Algorithm 3.4).

Algorithm 3.4: Decide whether to partial map or skip the node based on node type

```
   Data: netlist
   Result: partially mapped netlist
1  foreach node ∈ netlist do
2      switch node.type do
3          case arithmetic do
4              mark node to be mapped;
5          case memory do
6              mark node to be mapped;
7          otherwise do
8              mark node to be skipped;
```

The netlist is traversed using a depth-first search (DFS) approach from the top input nodes and for every node being visited, if the node is an arithmetic type (add, subtract, and multiply) or a memory type, it is marked to be mapped later during the partial mapping stage. Otherwise the node is marked to be skipped and no operation will be performed on it. If a user needs to change whether a specific type should be mapped or skipped, the adjustment can be easily done as there is a list of types to be mapped and the others will be ignored.

3.6 Partial Mapping Algorithm

The partial mapping algorithm in Parmys is derived from the Odin-II partial mapping algorithm with slight modifications. The process starts with iterating over all nodes within the netlist. If a node is marked to be skipped in the skip/map stage, then the node is kept as is without being touched. This is useful when a type is defined as a black box and the function is unknown or when a cell is already par-

tially mapped and there is no benefit in performing the partial mapping again. The skip mechanism speeds up the partial mapping stage as many nodes are skipped and only a subset of nodes within the netlist are processed in the partial mapping stage (Algorithm 3.5).

In Odin-II, the partial mapper utilizes fine-grain primitives to implement the soft logic. Unlike Odin-II, Parmys uses Yosys compatible coarse-grain cells to implement the soft logic to preserve the optimization potential later within Yosys.

The nodes of arithmetic types (add, subtract, and multiply) and memory nodes are top candidates for the partial mapping. The arithmetic and memory nodes are marked to be processed in the skip/map stage. For an adder node, the partial mapper has to check if there are available hard blocks (hard adders) remaining in the target FPGA and if no suitable blocks are available then soft logic (coarse-grain primitives) is used to implement the node, otherwise, the node is implemented using the hard adder in the architecture and the available resources (remaining hard blocks) are updated to reflect the recent updates.

A subtraction can be rewritten as an addition using 2's complement. To increase resource utilization, FPGA architectures usually contain only hard adders and the subtraction is implemented using addition. During partial mapping, if a node is of subtract type, first it is converted to an addition with slight adjustments. Then the rest is the same as for an adder node. If there are available hard adders, the partial mapper implements the logic node using the hard blocks and if there are no useful resources in the target architecture, soft logic is used for the implementation.

For multiplication, the user can set a hyper parameter to control the mixing implementation. If a mixing hard/soft parameter is set, then the partial mapper only tries to map to a subset of the hard multipliers. For example if the user set the ratio to 0.25, then the partial mapper will try to utilize only 25 percent of the hard multipliers and soft logic is used to implement the rest of the multipliers. If no ratio

Algorithm 3.5: Partial mapping algorithm

```
   Data: netlist
   Data: hard blocks
   Result: partially mapped netlist
1  foreach node ∈ netlist do
2      switch node.type do
3          case ADD do
4              if hard adders are available then
5                  map to hard adders
6              else
7                  map to soft logic
8          case SUBTRACT do
9              if hard adders are available then
10                 map to hard adders
11             else
12                 map to soft logic
13         case MULTIPLY do
14             if hard multipliers are available then
15                 map to hard multipliers with respect to mixing hard/soft
                     constraints;
16             else
17                 map to soft logic;
18         case MEMORY do
19             if hard memory blocks are available then
20                 map to hard memory blocks;
21             else
22                 map to DFFs;
23         case SKIP do
24             keep node as is;
25         otherwise do
26             raise error;
```

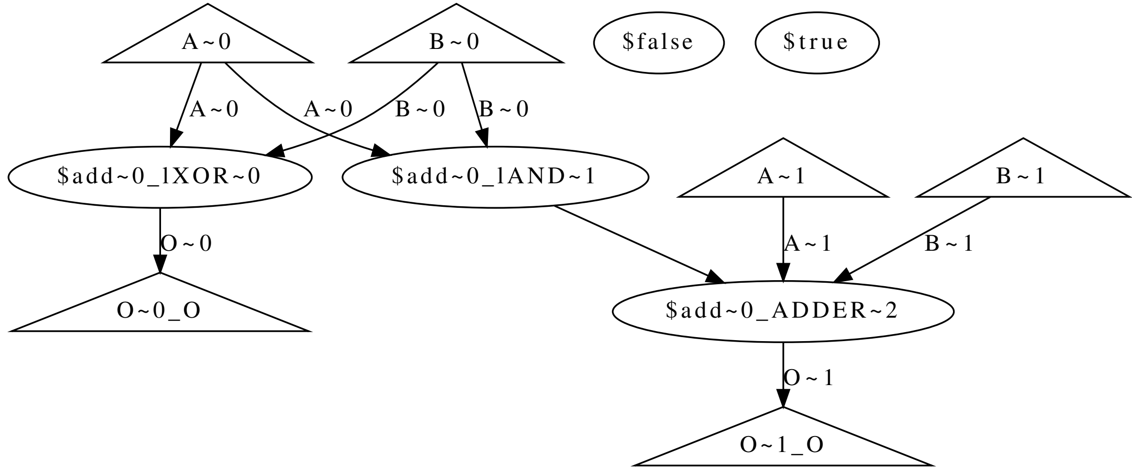

Figure 3.6: `add_2` netlist in the Odin-II space after partial mapping *(soft)*.

is set, then the partial mapper will try to map to all available hard multipliers within the target device.

For memory nodes, the same logic applies with slight modification. If there are free memory blocks within the target, the hard logic is used, otherwise, the memory functionality is implemented using DFFs within the architecture. Furthermore, If any new type is to be added for mapping during the partial mapping stage, the type first has to be marked not to be skipped during the skip/map stage, and the implementation algorithm has to be defined and injected to the partial mapping stage.

Figure 3.6 illustrates the partially mapped netlist. Since no architecture is provided, the circuit is mapped to soft logic.

3.7 Update

To this point, the netlist is mapped within the Odin-II space so now the Yosys design should be updated to reflect the partially mapped logic. The netlist in the Odin-II space is traversed recursively and every top input, top output, and internal nodes are back-transformed to the Yosys space as input wires, output wires, and cells. Also, if

there are any hard blocks, hard adders, or hard multipliers in the netlist, a black box module will be added to the design to support the transformed cell (Algorithm 3.6).

Algorithm 3.6: Update the design in the Yosys space

```
   Data: netlist
   Data: hard blocks
   Data: design
   Result: updated (partially mapped) design
 1 module ← new_module();
 2 foreach node ∈ netlist.input_nodes do
 3 |   port ← new_port();
 4 |   port.input ← true;
 5 |   add port to module;
 6 foreach node ∈ netlist.internal_nodes do
 7 |   cell ← create_cell(node.type);
 8 |   add cell to module;
 9 foreach node ∈ netlist.output_nodes do
10 |   port ← new_port();
11 |   port.output ← true;
12 |   add port to module;
13 add module to design as top module;
14 foreach hb ∈ hard blocks do
15 |   hb_module ← new_module();
16 |   setup ports and attributes for hard block;
17 |   hb_module.black_box ← true;
18 |   add hb_module to design;
```

Figure 3.7 visualizes the updated design in Yosys (the Yosys internal cell types, e.g., $reduce_xor and $reduce_and, are used as soft logic types).

3.8 Optimization

The netlist optimization is not a straightforward problem with an optimal solution. Synthesis is performed in multiple steps and no generic optimization approach is best suited at each stage. Optimization in Yosys is performed at different levels and each optimization algorithm targets a specific logic granularity. The Parmys plugin utilizes the Yosys optimization features before and after partial mapping to reach a

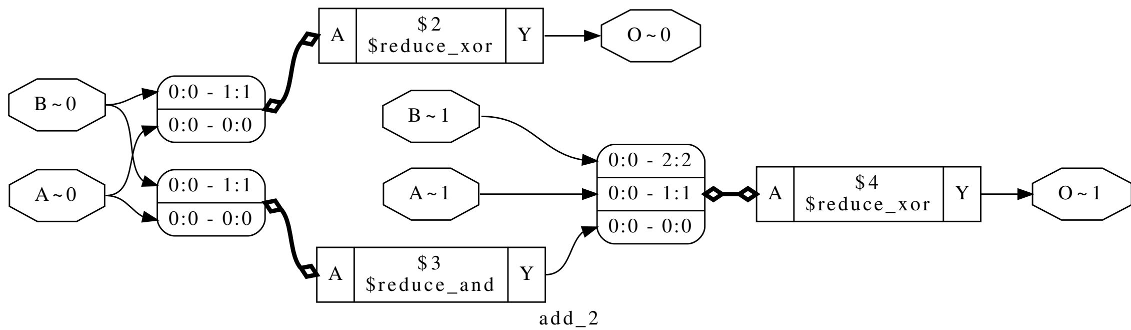

Figure 3.7: `add_2` updated design in the Yosys space after partial mapping *(soft)*. Since no architecture is provided all adders are mapped to soft logic using primitive types.

compact design to simplify the underlying circuit for the next stages of the synthesis pipeline. Additionally, the Parmys plugin is designed to postpone logic optimization as much as possible while performing the partial mapping so that Yosys can better optimize the circuit later.

3.9 FPGA Architecture File

If an FPGA architecture is provided, then the Parmys pass will try to (partially) map the circuit to available hard blocks. Figures 3.8 and 3.9 show the effect of architecture awareness, where hard adders are used instead of soft logic. The corresponding architecture file defines a single bit hard adder (Figure 3.10) for the details.

To reach maximum speed and efficiency and also to get more out of the Yosys optimizer, during partial mapping only certain types will be mapped and the rest, including other Yosys internal cell types, black box modules, and previously mapped logic, will be skipped.

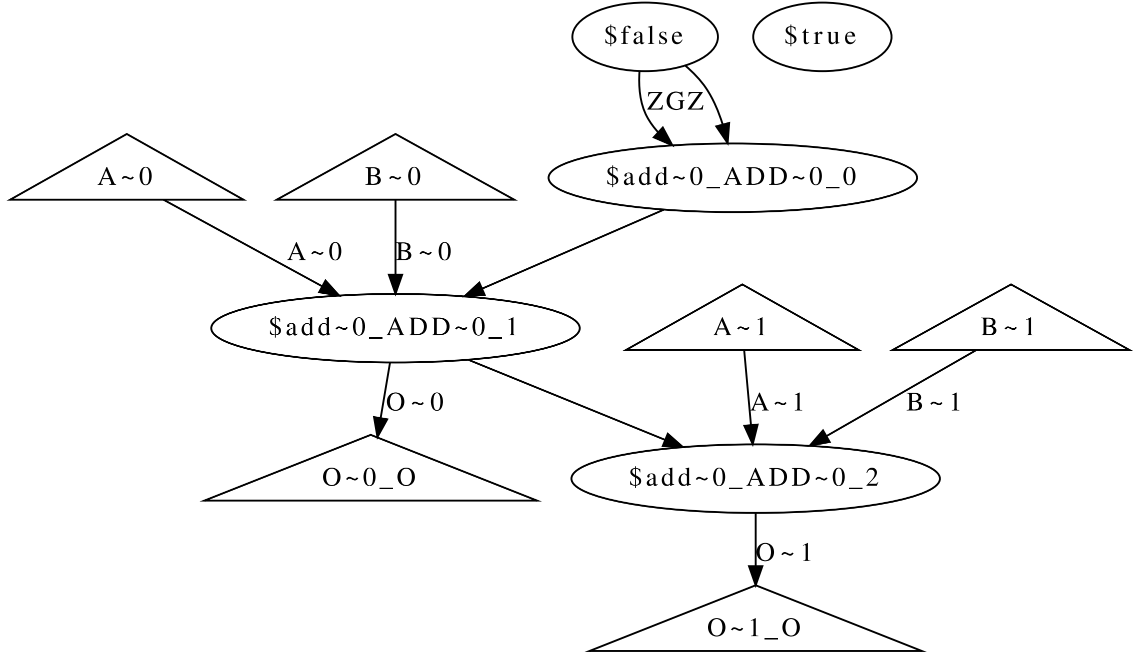

Figure 3.8: `add_2` netlist in the Odin-II space after partial mapping *(hard)*.

3.10 Parmys Information Flow

Figure 3.11 illustrates the flow of information within the Parmys pass. The RTLIL design is handed to the Parmys pass, then the transform stage translates the design to an equivalent netlist. Next the elements within the netlist are skipped or mapped to the target FPGA with respect to the parameters provided by the user. Finally the mapped netlist is used to update the RTLIL design. After or before the Parmys pass, the design may be processed by other passes for any purpose.

3.11 Verification by Unit Tests

Odin-II has a built-in simulator and numerous test cases were implemented for Odin-II as pairs of input-output vectors for each Verilog benchmark, including the standard VTR benchmark. Originally, Odin-II read the Verilog description and then output the partially mapped circuit after RTL synthesis as a BLIF file. Next, the BLIF

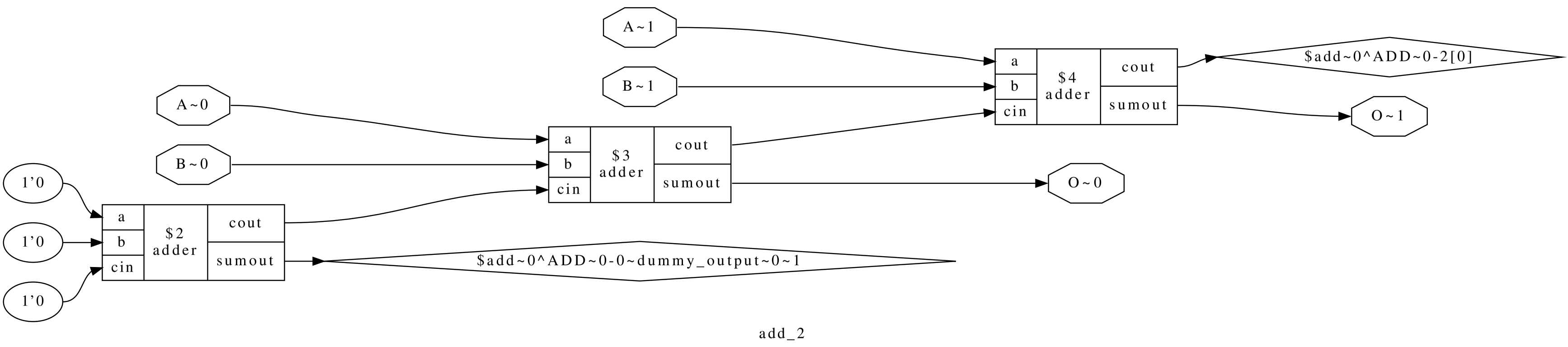

Figure 3.9: `add_2` updated design in the Yosys space after partial mapping *(hard)*.

```
1    .model adder
2    .inputs cin b a
3    .outputs sumout cout
4    .blackbox
5    .end
```

Figure 3.10: Definition of `adder` hard block in BLIF format.

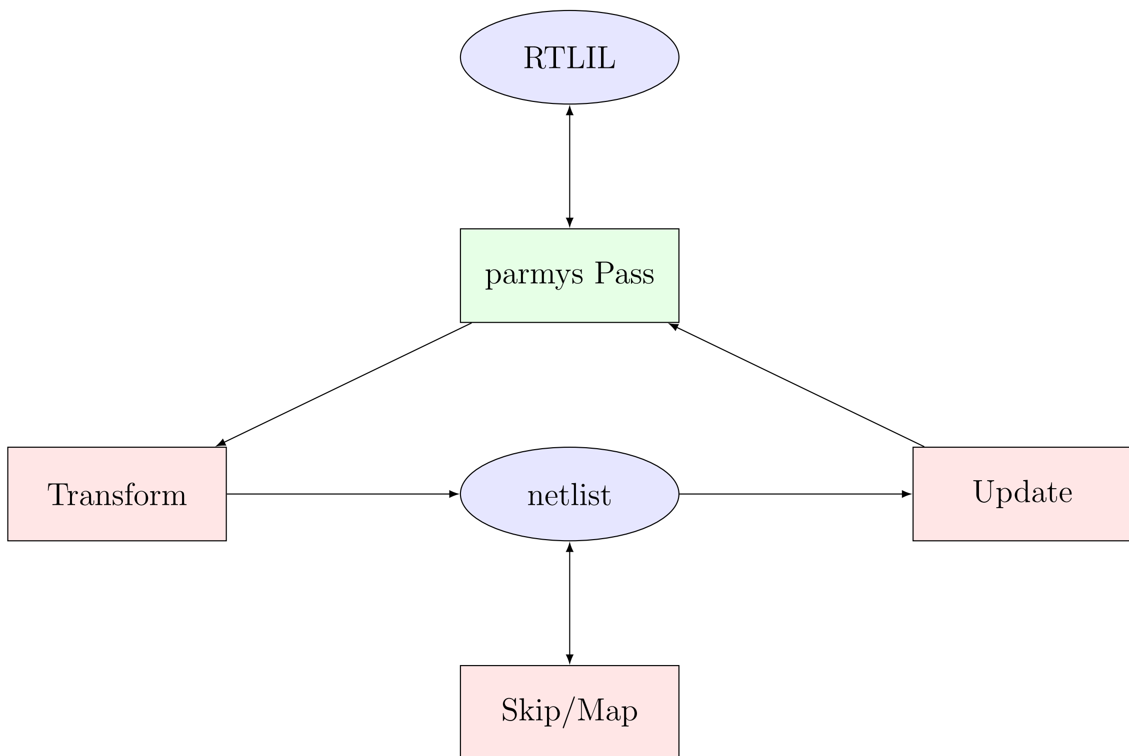

Figure 3.11: Final flow

output is interpreted by the BLIF reader and then the simulation process is started using the input vectors. Each row in the input vector determines the values for the circuit input pins at each clock cycle. Then the simulator computes the output of the circuit for each output pin with respect to the inputs and generates the outputs per each clock cycle. Finally, the generated outputs are compared with the expected outputs and they should match to pass the test.

The Odin-II test cases with necessary adjustments were utilized to verify the correctness of the partial mapper plugin. The procedure for verification in Parmys is almost the same as Odin-II. Each Verilog description is elaborated by Yosys and then it is partially mapped with Parmys and the output is saved as a BLIF file. The BLIF output file is then interpreted by the Odin-II BLIF reader and is simulated by the Odin-II simulator. For each clock cycle the values of the output pins are collected based on the values of the input pins. If the generated output matches the expected output the test is passed. Parmys passed all available test cases within the Odin-II source.

3.12 Passes

To get the best out of the partial mapping plug-in, a chain of Yosys passes were utilized before and after the Parmys pass. Figure 3.12 illustrates the chain of passes. The `plugin` pass loads and activates the Parmys plug-in into Yosys. The `parmys_arch` pass was implemented to load the target architecture into the Yosys design.

The `read_verilog` pass triggers the Verilog frontend to read the input circuit(s). The `opt` pass runs a group of optimization passes including `opt_clean`, `opt_expr`, and `opt_muxtree` to perform full or partial optimization based on the parameters. The `fsm` pass runs a family of passes to extract and optimize finite-state-machines from the design. The `proc` pass is utilized to translate algorithmic logic to RTL

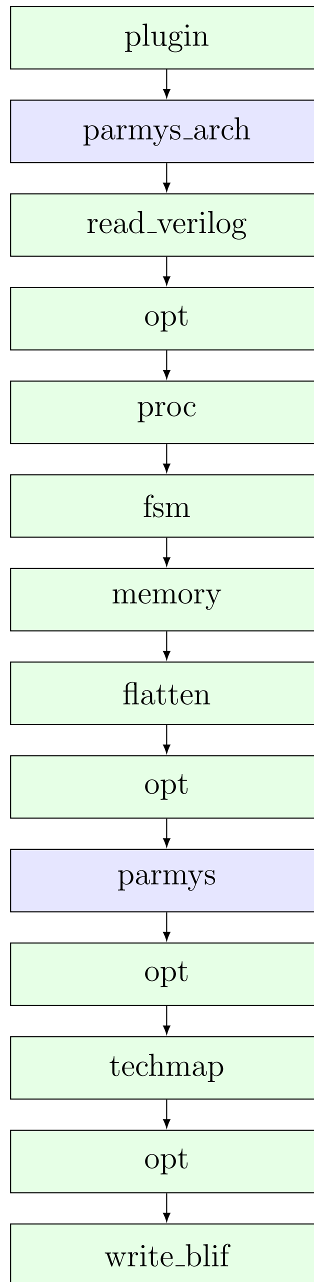

Figure 3.12: Chain of passes (green: Yosys standard passes, blue: Parmys passes)

elements. The `memory` pass converts high-level memory logic to RTL memory components. The `flatten` pass replaces each module instance within the top module with its implementation and at the end only one module is present within the design. The `parmys` pass performs partial mapping to map adders (subtractors), multipliers, memories, and custom hard-blocks to available components within the target architecture. The `techmap` pass maps the remaining coarse-grained logic to fine-grained-logic. The `write_blif` pass triggers the BLIF backend to export the design to a BLIF file.

The `parmys_arch` and `parmys` passes were implemented as part of the Parmys plug-in and other passes are standard features of Yosys. The proposed order is based on a series of experiments and best practices suggested by the Yosys developers. The order of passes may be altered by the final user to tweak performance or to use other frontends and backends as needed.

Chapter 4

Parmys Frontend

The *Transform* stage of the Parmys plug-in creates an equivalent Odin-II compatible netlist from the Yosys Open SYnthesis Suite (Yosys) design and with a few modifications the output may be passed to the Verilog to Routing (VTR) flow. In other words Yosys could serve as the frontend of the VTR flow instead of Odin-II, taking advantage of the *Transform* phase while utilizing the target Field Programmable Gate Array (FPGA) architecture. To extend this work, a new standalone frontend has been added to the VTR flow by integrating the Parmys plug-in and the Yosys elaborator.

4.1 Idea

The Parmys plug-in provides VTR-style architectural awareness for the Yosys users. The Odin-II frontend in VTR flow lacks support for the Verilog-2005 standard and the optimization in Odin-II is not as powerful as other Hardware Description Language (HDL) elaborators such as Yosys. With intelligent partial mapping features being accessible through the Parmys plug-in in Yosys, the opportunity is created for introducing a new frontend by combining Yosys and the Parmys plug-in.

4.2 Conventional VTR Flow

The conventional VTR flow consists of three main components: Odin-II as the elaborator and partial mapper, ABC for logic optimization and physical mapping, and VPR for packing, routing, and placement (Figure 2.5).

Although Odin-II provides intelligent partial mapping for the VTR flow, lack of support for new hardware description standards and outdated optimization approaches make Odin-II less attractive for the open-source community.

4.3 New VTR Flow

The Odin-II frontend in the VTR flow is replaced with the Parmys frontend to perform logic elaboration by Yosys and partial mapping by Parmys plug-in (Figure 4.1). The Parmys frontend integrates modern HDL elaboration features from Yosys and intelligent partial mapping features from Odin-II by embedding the Parmys plug-in into Yosys and utilizing it as a new standalone frontend within the VTR flow.

4.4 Verification by Regression Tests

The VTR flow utilizes regression tests to ensure that with every development, the quality of results remain stable. After adding the Parmys frontend to the VTR flow, previous regression tests were duplicated and adjusted to utilize Yosys as the elaborator and Parmys as the partial mapper. Quality of Result (QoR) metrics were recorded for each benchmark and new regression tests were added to the continuous integration pipeline.

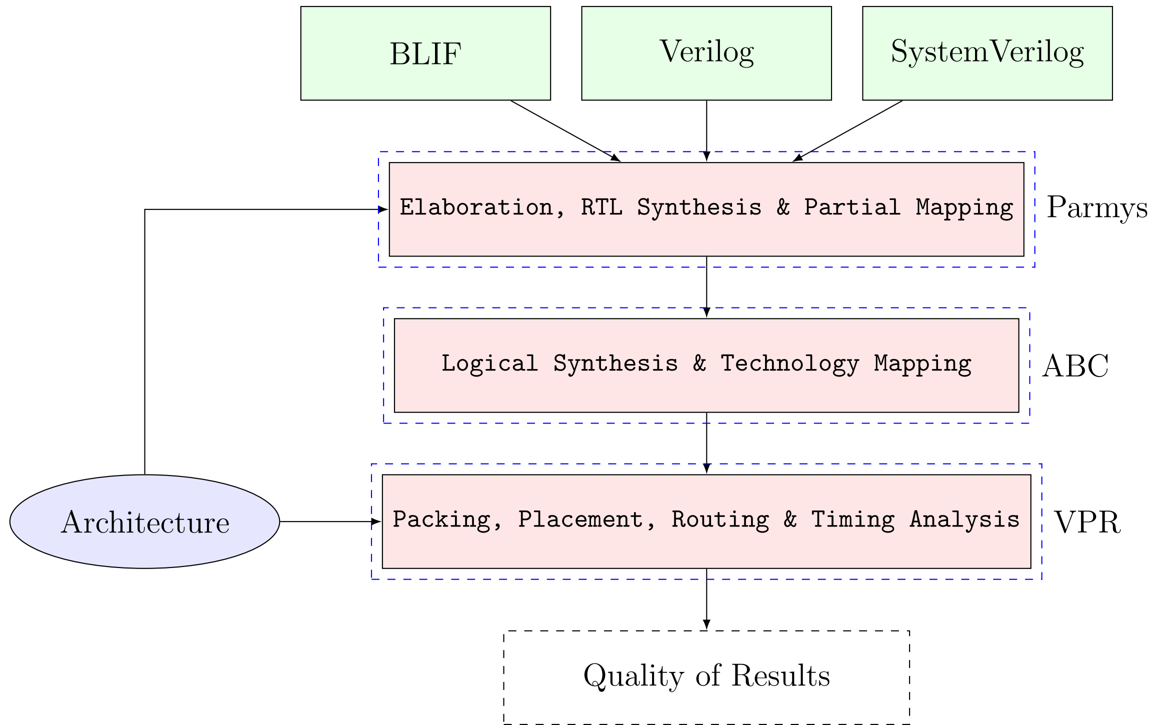

Figure 4.1: The new VTR CAD flow (Parmys, ABC, VPR)

4.5 New Regression Tests

Elaboration by Yosys expands Verilog support for the VTR flow frontend. Many open source benchmarks that previously were not supported by the VTR flow due to a lack of support for Verilog are now supported with the Parmys frontend. Three sets of new regression benchmarks were added to the VTR flow: `ultra embedded`, `VexRiscv`, and `free cores` [12]. Table 4.1 lists test cases within each benchmark. Adding new benchmarks helped us to discover and address a few memory mapping corner cases in the partial mapping algorithm.

4.6 Odin-II Deprecation

Formerly, if VTR users intended to utilize other frontends for better Verilog support or to input other HDL formats, they lost partial mapping features provided by Odin-II or they had to handcraft the logic inference and partial mapping. Now the Parmys

Table 4.1: Benchmarks

ultra embedded	VexRiscv	free cores
enet_core [7]	VexRiscvFullNoMmuMaxPerf	aes_cipher
mmc_core [10]	VexRiscvFullNoMmuNoCache	aes_inv_cipher
soc_core [4]	VexRiscvFullNoMmu	8051 [1]
uriscv_core [13]	VexRiscvFull	ethmac [6]
usb_uart_core [14]	VexRiscvLinuxBalancedSmp	mips_16 [9]
	VexRiscvLinuxBalanced	xtea [16]
	VexRiscvNoCacheNoMmuMaxPerf	
	VexRiscvSecure	
	VexRiscvSmallAndProductiveICache	
	VexRiscvSmallAndProductive	
	VexRiscvSmallestNoCsr	
	VexRiscvSmallest	

frontend not only provides more robust and extended HDL elaboration support, but also automates the intelligent partial mapping from logic inference to binding and hard or soft logic trade-offs.

As of commit `5845ee0`[1] (December 16, 2022) the Parmys code is integrated into the main branch of the VTR repository[2] and as of commit `2a29e57`[3] (February 23, 2023) the Parmys frontend is the default frontend in the VTR flow[4] and this provides the opportunity to deprecate Odin-II in a future release. Yosys+Odin-II was removed from the VTR repository as of commit `f669015`[5] (January 24, 2023) due to effectiveness and efficiency issues[6].

[1] https://github.com/verilog-to-routing/vtr-verilog-to-routing/commit/5845ee0
[2] https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/2215
[3] https://github.com/verilog-to-routing/vtr-verilog-to-routing/commit/2a29e57
[4] https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/2234
[5] https://github.com/verilog-to-routing/vtr-verilog-to-routing/commit/f669015
[6] https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/2232

Chapter 5

Results and Analysis

As a proof of concept, a sample circuit (which will be referred to as the `complex` circuit) is designed to show how the Parmys plug-in maps logical elements to available hard blocks within the target architecture. Moreover, the performance of the Parmys plug-in and Parmys frontend were evaluated using the Verilog to Routing (VTR) benchmarks and the Koios benchmarks and the results and analysis are documented in this chapter. The experiments represented in this chapter were designed to evaluate the quality of results in an end-to-end Electronic Design Automation (EDA) flow.

5.1 Proof of Concept

Figures 5.1, 5.2, and 5.3 show the Verilog description, circuit visualization and the equivalent netlist of the `complex` circuit. As Figure 5.2 shows, the `complex` circuit consists of two multipliers, a Digital Signal Processing (DSP) black box and a logical cell. As per the partial mapping algorithm, due to the availability of the hard blocks in the Field Programmable Gate Array (FPGA) architecture, the `$mul` cells could be mapped to hard multipliers but the black box and logical cell will be skipped during partial mapping. The `mults_ratio` parameter is set to 0.5 so half of the multiply

```verilog
`define WIDTH 2

module complex(A, B, C, D, O);

input [1:0] A,B,C,D;
output O;
wire [3:0] M,N;
wire [1:0] P;

assign M = A * B;
assign N = C * D;

dsp box(.R(M), .S(N), .T(P));

assign O = ^P;

endmodule

(* black_box *)
module dsp(R, S, T);
input [3:0] R,S;
output [1:0] T;
endmodule
```

Figure 5.1: Description of `complex` circuit in Verilog format.

cells should be mapped to the hard multipliers and the other half should be exploded into soft logic (Figure 5.4).

After updating the Yosys design (Figure 5.5), Yosys compatible soft logic is used to represent the soft-mapped multiplier, and the hard-mapped multiplier is instantiated as `mult_2_2_4`, which is the VTR standard naming for hard multipliers (Figure 5.6). The black box and `$reduce_xor` logical cell were skipped and kept intact.

5.2 Parmys Frontend in the VTR flow

The standard VTR benchmarks [45, 54] are utilized to see how the Parmys plugin performs in Yosys. These benchmarks include a set of medium-sized circuits and the

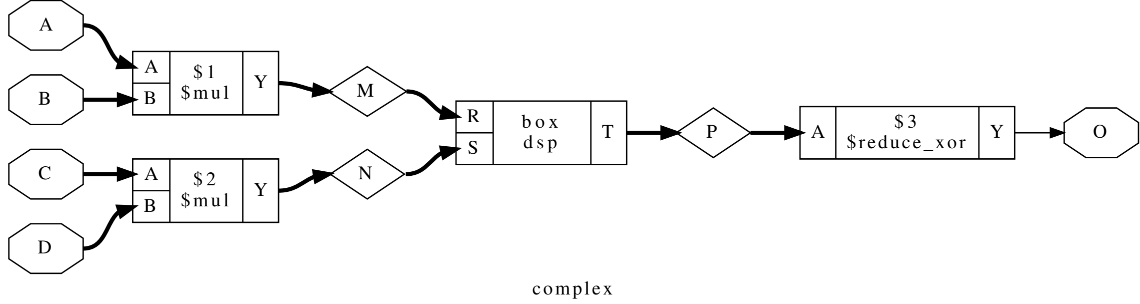

Figure 5.2: `complex` circuit visualization in the Yosys space.

Table 5.1: VTR flagship architecture characteristics

Metric	Value
Process Technology	40nm
LUT Size	6 (fracturable)
BLEs per Logic Block	10
Adder Bits per Logic Block	20
Memory Block Size	32kb
DSP Block Multiplier Size	36x36 (fracturable)
Switchblock Type	Wilton
Wiretype Length	4

target FPGA is known as the VTR flagship architecture. The characteristics of the VTR Flagship architecture are listed in Table 5.1.

To better understand the effect of the Parmys plugin in an end-to-end flow, Yosys and the Parmys partial mapper are used as the VTR frontend. VTR, Yosys, and Parmys were compiled with GCC9 and results were collected on Ubuntu 20.04.4 LTS (GNU/Linux 5.4.0-135-generic x86_64) operating system in an isolated test environment (no other users or background tasks). Each experiment was performed with 3 random seeds and the geometric mean of runs was reported. To run the experiments 32 Intel Xeon CPU E7520 processors running at 1.87GHz with 128GB of memory were used.

Each Verilog circuit within the VTR benchmark is first elaborated using Yosys, then a series of Yosys internal passes are applied to the design to replace behavioural structures with RTL elements and optimize the circuit. Then the Parmys pass is

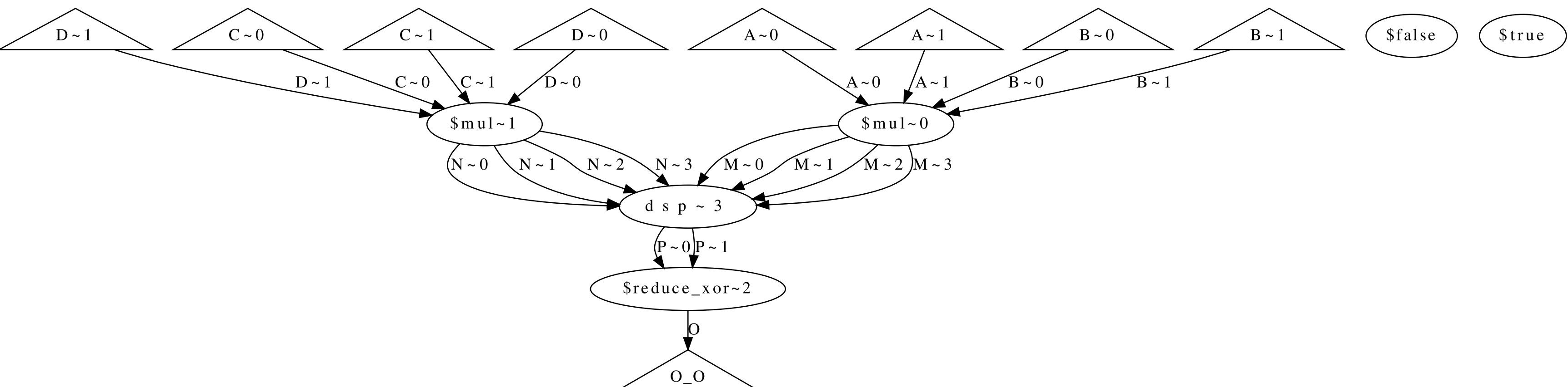

Figure 5.3: `complex` equivalent netlist in the Odin-II space after transform before partial mapping.

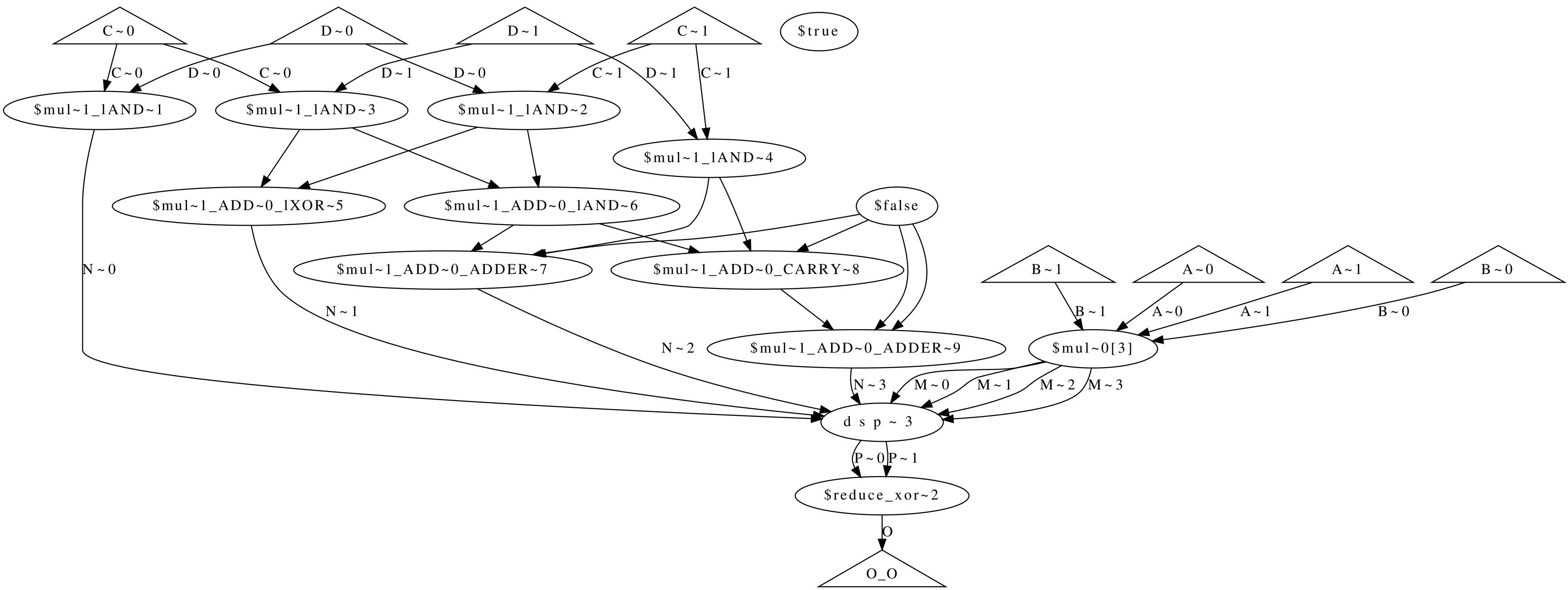

Figure 5.4: `complex` netlist in the Odin-II space after partial mapping.

Figure 5.5: `complex` updated design in the Yosys space.

```
.model mult_2_2_4
.inputs a[0] a[1] b[0] b[1]
.outputs out[0] out[1] out[2] out[3]
.blackbox
.end
```

Figure 5.6: Definition of mult_2_2_4 hard block in BLIF format.

called to perform partial mapping according to the target architecture, and then a series of optimization and logic mappings are performed to convert all the remaining Yosys internal types to primitives. Next the output is passed to ABC and VPR stages and at the end of the flow the Quality of Result (QoR) metrics are collected. According to the VTR documentation [30], the key QoR metrics are:

- Pre-Packed Blocks: The number of primitive netlist blocks after technology mapping before packing.
- Post-Packed Blocks: The number of Clustered Blocks after packing.
- Device Grid Tiles: The FPGA size.
- Min Channel Width: The minimum routable channel width.
- Wire Length: The critical path routed wire length.
- Critical Path Delay: The maximum cumulative delay

To better investigate the performance of the underlying synthesis tool, it is suggested to only consider the circuits with more than (or equal to) 10,000 primitives [49]. Therefore, circuits with less than 10,000 primitives were excluded from the VTR benchmarks. Each reported metric is averaged for all of the qualifying circuits withing the benchmark.

Figure 5.7 shows the synthesis time in the frontend increased by 167% (from 16.5 seconds to 44 seconds) while ABC time decreased by 63% (from 198 seconds to 73 seconds) and the total VTR flow elapsed time shows around 30% improvement

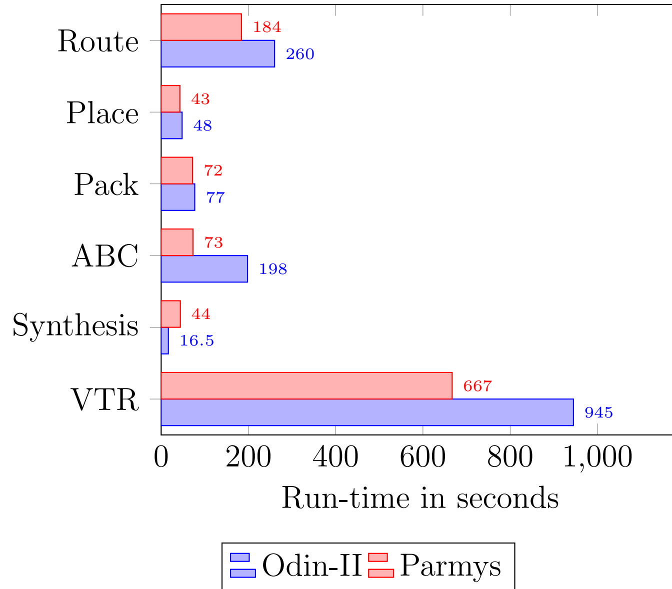

Figure 5.7: Run-time comparison for the VTR flow utilizing the Odin-II synthesizer versus the Parmys synthesizer.

(decreased from 945 seconds to 667 seconds). The jump in the synthesis time is mainly related to the fact that Yosys optimizes the design using various algorithms at different levels. This proves effective because the design is more optimized and less effort is needed in the ABC and VPR stages.

Figure 5.8 shows the maximum memory usage in the frontend (the synthesis stage) increased by 51% (from 179 MB to 271 MB) while ABC memory decreased by 8.5% (from 117 MB to 107 MB). The VPR stage consumes 16% less memory with the Parmys frontend (decreased from 531 MB to 446 MB) and the total VTR flow maximum memory shows around 9% improvement (decreased from 515 MB to 468 MB). Yosys has larger source code in comparison to Odin-II and numerous components are loaded into memory mainly accounting for the jump in synthesis memory. However, the smaller output of Yosys requires less memory within the ABC and VPR stages.

Figure 5.9 shows the number of CLBs and multipliers decreased by 8% (from 1,462 to 1,352) and 11% (from 8.63 to 8.01) respectively while the number of memories remained the same on average (∼4.5). The `opt_expr` pass in Yosys performs constant folding. That means, if a cell of a specific type is connected to constant (or partially

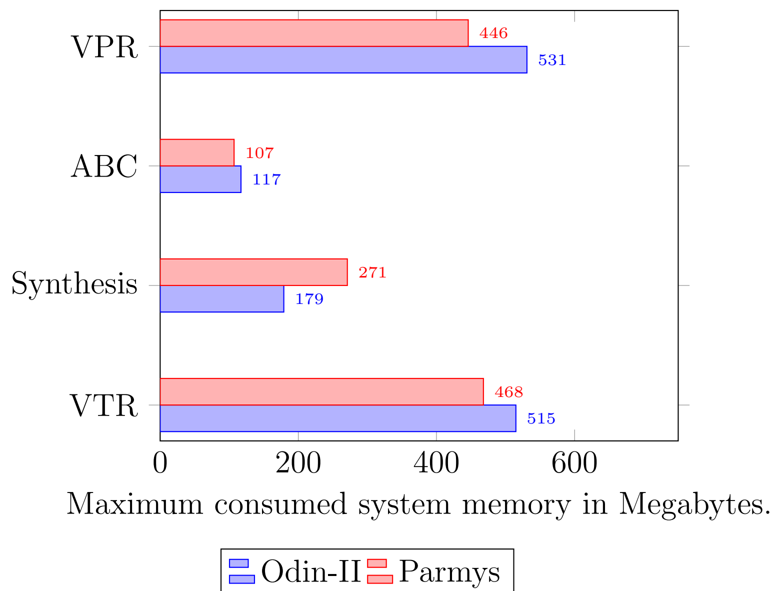

Figure 5.8: Memory comparison for the VTR flow utilizing the Odin-II synthesizer versus the Parmys synthesizer.

constant) inputs, then the cell may be replaced with the constant values that are driving the cell inputs. For example consider a simple two input `OR` gate, if both of the inputs are driven by 1, then the cell can be replaced with the signal 1. The `opt_expr` pass is called once when the optimization pass starts, since there is no point in running it multiple times.

The `opt_muxtree` pass simplifies nested multiplexers by removing unreachable conditions and reducing the depth of a tree of multiplexers. The `opt_clean` pass discovers unused cells and wires and removes them from the circuit. The `opt_expr`, `opt_muxtree`, and `opt_clean` passes are repeated until no further improvements are applicable to the underlying circuit, accounting for the decrease in the number of the hard blocks and CLBs.

Figure 5.10 shows the number of pre-packed and post-packed blocks decreased by 11% (from 30,303 to 26,969) and 7% (from 1,852 to 1,729) respectively, accounting for the improvements in the pack (7%—from 77 seconds to 72 seconds), place (10%—from 48 seconds to 43 seconds), and route (30%—from 260 seconds to 184 seconds) times (refer to Figure 5.7 for more details). The Parmys plugin utilizes the

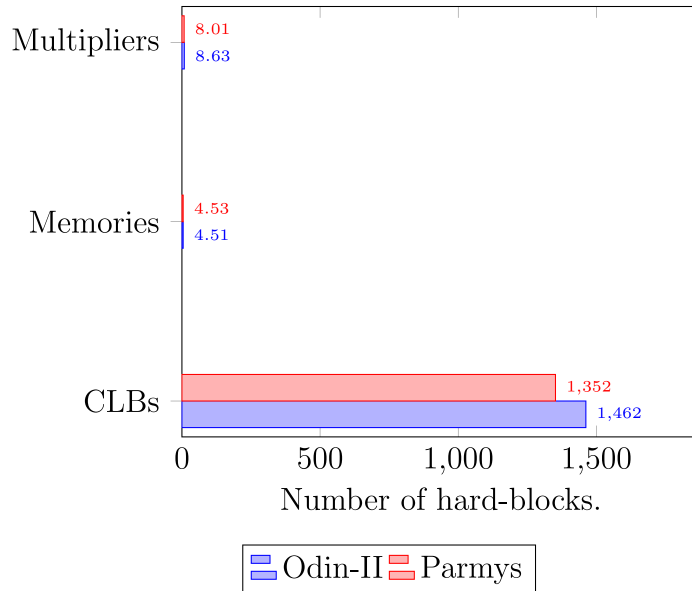

Figure 5.9: Number of hard-blocks comparison for the VTR flow utilizing the Odin-II synthesizer versus the Parmys synthesizer.

optimization passes at different stages of the synthesis flow. The initial circuit after Hardware Description Language (HDL) elaboration is a coarse-grained design and the optimization passes are focused on the coarse-grain components. As the design proceeds through the synthesis pipeline, the elements become more fine-grained and optimization passes work on the details locally. With the design being optimized at both local and global levels, the ABC pass has much less to optimize and is more focused on physical mapping. Less logic is equal to a smaller design, or a lower number of pre-packed blocks. As the input circuit to VPR is smaller, the output tends to be smaller, accounting for a lower number of the post-packed blocks.

Figure 5.10 shows the minimum channel width remained almost the same with a 3% increase (from 85.6 MHz to 88.7 MHz), which is acceptable given the fact that well-known commercial FPGAs are programmable with a channel width up to 300 MHz [53], whereas the absolute value for the minimum channel width for the VTR benchmarks using the Parmys plugin is around 90 MHz (with a minimum of 56 MHz for the `stereovision0` circuit and a maximum of 144 MHz for the `mcml` circuit). The critical path routed wire length is improved by 5% (from 212,997 to

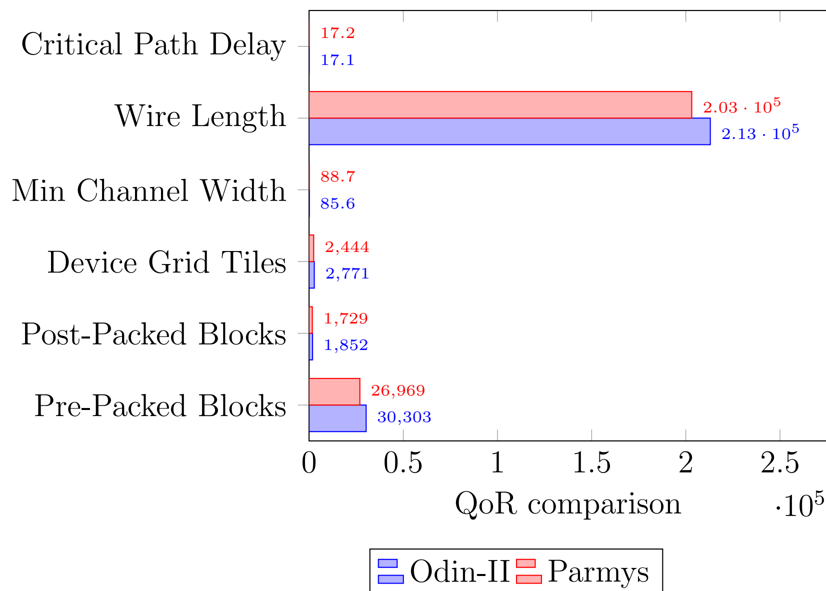

Figure 5.10: QoR comparison for the VTR flow utilizing the Odin-II synthesizer versus the Parmys synthesizer.

203,136) and critical path delay did not change (∼17 clock cycles).

To further validate the process, the QoR metrics were collected for three other benchmarks. The Koios [20] benchmarks are medium-sized to large-sized deep learning benchmarks, with three standard experiments utilizing the Koios benchmarks in the VTR flow. The `default` Koios task targets an FPGA architecture that includes user-defined hard blocks to perform specific neural network computations. The `no_hard_block` Koios task only uses generic VTR hard blocks and no custom hard blocks. The `multi_arch` Koios task utilizes 11 different targets for only the circuit named `conv_layer.v` to study the effect of the FPGA architecture on the QoR metrics.

Table 5.2 summarize the QoR metrics for the Koios tasks. Results are relative to the baseline VTR settings and the geometric mean of all the circuits within each benchmark is reported. Minimum channel width, critical path routed wire length, and critical path delay were not reported for Koios tasks. The statistics for multipliers are reported if hard multipliers are present within the target architecture. The first column (labelled default) shows a 7% increase for the synthesis time, while

	koios*		
	default	no_hard_block	multi_arch
VTR Flow Elapsed Time	0.53	0.47	0.35
Synthesis Time	1.07	1.54	1.62
Logic Depth	0.95	0.93	1.00
ABC Time	0.29	0.26	0.26
CLBs	0.89	0.81	0.76
Memories	0.99	0.95	1.00
Multipliers	-†	0.95	-†
Pre-Packed Blocks	0.90	0.83	0.80
Post-Packed Blocks	0.92	0.86	0.80
Device Grid Tiles	0.97	0.98	0.88
Pack Time	0.62	0.55	0.28
Place Time	0.59	0.52	0.54

Table 5.2: Koios benchmarks using Yosys as the elaborator and the Parmys plug-in as the partial mapper in the VTR flow.

* The channel width is set to 300 by default, therefore VPR did not report the min channel width, wire length, and critical path delay metrics for the Koios benchmarks.

† The target architecture uses custom defined multipliers and generic multipliers are not utilized.

the whole flow time is reduced by 47%. The improvements in ABC and VPR times account for the major drop in the end-to-end elapsed time. The increase in the synthesis time for the Koios benchmarks is less than the VTR benchmarks, since the Koios benchmarks utilize a higher ratio of hard blocks in comparison to the VTR benchmarks. The synthesis time in the second column (labelled no_hard_block) supports this observation, since this metric is collected with no hard blocks and the synthesis takes longer to optimize when hard blocks are replaced with soft logic implementation.

Results in the last column (labelled multi_arch) reveals that the target architecture has huge impacts on the final results as metrics are collected utilizing 11 different FPGA architectures and almost all metrics are better than the metrics for the default experimental setup.

The logic depth for the Koios benchmarks showed a better improvement in comparison to the VTR benchmarks, accounting for the major drop in the ABC time.

The Koios benchmarks are heavily pipelined [20], therefore the logic in the Koios benchmarks is deeper than the VTR benchmarks on average. The optimization algorithms create a better output when the logic tree is deeper or larger, therefore the Yosys optimizer can produce a more optimized netlist for the Koios benchmarks when utilizing the Parmys plugin, relative to the conventional VTR flow.

Results indicated that all key QoR metrics for the VTR benchmarks with the flagship FPGA architecture and Koios benchmarks were improved. Compared to the VTR benchmarks, the Koios benchmarks performed better for every QoR metric. The main reason is that circuits within the VTR benchmarks are medium-sized while the Koios benchmarks contain mainly medium-sized to large-sized designs and as the size of the design increases, the ratio of the warm-up time to the run-time for each stage of the VTR flow is reduced. Furthermore, the larger design provides better optimization opportunity within Yosys, and as the design gets more optimized, ABC and VPR can better map, pack, and route.

With each development in the VTR synthesis pipeline, the amount of the improvement is expected to be less than 3% for every QoR metric [30]. The results showed a high rate of performance for the Parmys plugin in comparison to the conventional VTR settings. A 30% to 50% improvement in the end-to-end flow time, and around 10% in the final circuit size is considered a large impact to the synthesis speed and the area efficiency for an EDA tool.

Chapter 6

Conclusions and Future Work

This thesis proposes the partial mapping plug-in for Yosys Open SYnthesis Suite (Yosys), the Parmys plug-in, and a new standalone frontend for Verilog to Routing (VTR), the Parmys frontend.

Extracting the Odin-II partial mapper and integrating it within the Yosys synthesis tool brings VTR-style Field Programmable Gate Array (FPGA) architecture awareness to Yosys. As a result, all the useful partial mapping features (inference, binding, and hard/soft logic trade-offs) from Odin-II are accessible through the Parmys pass in Yosys, where a real world or theoretical FPGA architecture can be used to preallocate specific circuit elements to hard blocks or even explode them to soft logic.

With the results of this research, Yosys users can benefit from a wide range of custom or industry-defined FPGA architectures in their synthesis flow. VTR users can also benefit from using Yosys as the frontend for elaboration, synthesis, and partial mapping without worrying about the FPGA architecture.

In addition to providing the new features in both Yosys and the VTR flow, the results demonstrated that the combination of the Parmys plug-in, the Yosys elaborator, ABC, and VPR can perform better than conventional VTR flow for almost every QoR metric.

6.1 Future Work

Odin-II is designed and implemented to serve as the Verilog frontend and also the partial mapper in the VTR flow, so it is heavily based on the VTR internal libraries, structures and standards. This work is a fundamental step toward extracting the Odin-II partial mapper and making it available to tools outside the VTR flow.

Unit tests were used to verify the output of the Parmys plugin and utilized regression tests to verify the integration of the Parmys frontend within the VTR flow. A possible future direction to extend this work would be to add formal verification to the VTR flow using Yosys formal verification and simulation features to make sure that with each development the logic synthesis remains valid.

Another potential area of research would be to provide alternative approaches when choosing among hard blocks to implement the logic. The approach taken to utilize the hard blocks can heavily affect the implementation performance. There are research opportunities to use artificial intelligence to decide among different implementation approaches. This is not limited to deciding whether to map to hard blocks or explode to soft logic, but also includes the selection among various available hard blocks (e.g., hard adders with different lengths).

Bibliography

[1] *8051 cores acccumulator*, https://opencores.org/projects/8051, (Accessed: 15 August 2023).

[2] *ABC: A system for sequential synthesis and verification*, https://people.eecs.berkeley.edu/~alanmi/a.

[3] *Advanced Micro Devices Incorporated*, https://www.amd.com/, (Accessed: 15 August 2023).

[4] *Basic Peripheral SoC*, http://github.com/ultraembedded/core_soc, (Accessed: 15 August 2023).

[5] *Berkeley Logic Interchange Format (BLIF)*, https://course.ece.cmu.edu/~ee760/760docs/blif.pdf.

[6] *Ethernet IP core project*, https://opencores.org/projects/ethmac, (Accessed: 15 August 2023).

[7] *Ethernet MAC 10/100 Mbps*, https://github.com/ultraembedded/core_enet, (Accessed: 15 August 2023).

[8] *Intel Corporation*, https://www.intel.com/, (Accessed: 15 August 2023).

[9] *MIPS 16 core*, https://github.com/freecores/mips_16, (Accessed: 15 August 2023).

[10] *MMC (and derivative standards) Host Controller*, https://github.com/ultraembedded/core_mmc, (Accessed: 15 August 2023).

[11] *MVSIS: Logic Synthesis and Verification*, https://ptolemy.berkeley.edu/projects/embedded/mvsis/.

[12] *Online community for the development of gateware IP (Intellectual Properties) Cores*, https://opencores.org/, (Accessed: 15 August 2023).

[13] *uriscv - Another tiny RISC-V implementation*, https://github.com/ultraembedded/core_uriscv, (Accessed: 15 August 2023).

[14] *USB Serial Port Device (USB-CDC)*, https://github.com/ultraembedded/core_usb_uart, (Accessed: 15 August 2023).

[15] *Xilinx Incorporated*, https://www.xilinx.com/, (Accessed: 15 August 2023).

[16] *XTEA IP Core*, https://opencores.org/projects/xteacore, (Accessed: 15 August 2023).

[17] *Technology Mapping*, pp. 505–521, Springer US, Boston, MA, 1996.

[18] *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) (2009), 1–640.

[19] Mariem Abid, *System-Level Hardwa Synthesis of Dataflow Programs with HEVC as Study Use Case*, Thesis, INSA de Rennes ; École nationale d'ingénieurs de Sfax (Tunisie), 2016.

[20] Aman Arora, Andrew Boutros, Daniel Rauch, Aishwarya Rajen, Aatman Borda, Seyed Alireza Damghani, Samidh Mehta, Sangram Kate, Pragnesh Patel, Kenneth B. Kent, Vaughn Betz, and Lizy K. John, *Koios: A Deep Learning*

Benchmark Suite for FPGA Architecture and CAD Research, 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), 2021, pp. 355–362.

[21] Vaughn Betz and Jonathan Rose, *VPR: a new packing, placement and routing tool for FPGA research*, Field-Programmable Logic and Applications (Berlin, Heidelberg) (Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner, eds.), Springer Berlin Heidelberg, 1997, pp. 213–222.

[22] Armin Biere, *The AIGER And-Inverter Graph (AIG) Format Version 20071012*, Tech. Report 07/1, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2007.

[23] Armin Biere, Keijo Heljanko, and Siert Wieringa, *AIGER 1.9 and beyond*, Tech. Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

[24] R.K. Brayton, G.D. Hachtel, and A.L. Sangiovanni-Vincentelli, *Multilevel logic synthesis*, Proceedings of the IEEE **78** (1990), no. 2, 264–300.

[25] Robert Brayton and Alan Mishchenko, *ABC: An Academic Industrial-Strength Verification Tool*, Computer Aided Verification (Berlin, Heidelberg) (Tayssir Touili, Byron Cook, and Paul Jackson, eds.), Springer Berlin Heidelberg, 2010, pp. 24–40.

[26] Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers **C-35** (1986), no. 8, 677–691.

[27] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski, *LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems*, Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable

Gate Arrays (New York, NY, USA), FPGA '11, Association for Computing Machinery, 2011, p. 33–36.

[28] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin, *GAUT: A High-Level Synthesis Tool for DSP Applications*, pp. 147–169, Springer Netherlands, Dordrecht, 2008.

[29] S.A. Damghani, *Yosys+Odin-II: The Odin-II Partial Mapper with Yosys Coarse-grained Netlists in VTR*, Master's thesis, University of New Brunswick, 2021.

[30] VTR developers, *Verilog To Routing Documentation*, `https://docs.verilogtorouting.org/`.

[31] Umer Farooq, Zied Marrakchi, and Habib Mehrez, *FPGA Architectures: An Overview*, pp. 7–48, Springer New York, New York, NY, 2012.

[32] D. Gajski and R. Kuhn, *New VLSI Tools*, Computer **16** (1983), no. 12, 11–14.

[33] D.D. Gajski and L. Ramachandran, *Introduction to high-level synthesis*, IEEE Design & Test of Computers **11** (1994), no. 4, 44–54.

[34] Ian Grout, *CHAPTER 2 - Electronic Systems Design*, Digital Systems Design with FPGAs and CPLDs, Newnes, Burlington, 2008, pp. 43–121.

[35] ———, *CHAPTER 6 - Introduction to Digital Logic Design with VHDL*, Digital Systems Design with FPGAs and CPLDs, Newnes, Burlington, 2008, pp. 333–474.

[36] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, *SPARK: a high-level synthesis framework for applying parallelizing compiler transformations*, 16th International Conference on VLSI Design, 2003. Proceedings., 2003, pp. 461–466.

[37] Eddie Hung, *Mind the (synthesis) gap: Examining where academic FPGA tools lag behind industry*, 2015 25th International Conference on Field Programmable Logic and Applications (FPL), 2015, pp. 1–4.

[38] P. Jamieson and J. Rose, *A Verilog RTL synthesis tool for heterogeneous FPGAs*, International Conference on Field Programmable Logic and Applications, 2005., 2005, pp. 305–310.

[39] Peter Jamieson, Kenneth B. Kent, Farnaz Gharibian, and Lesley Shannon, *Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research*, 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, 2010, pp. 149–156.

[40] Georgiy Krylov, Jean-Philippe Legault, and Kenneth B. Kent, *Hard and Soft Logic Trade-offs for Multipliers in VTR*, 2020 23rd Euromicro Conference on Digital System Design (DSD), 2020, pp. 40–43.

[41] A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. Ganai, *Robust Boolean reasoning for equivalence checking and functional property verification*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **21** (2002), no. 12, 1377–1394.

[42] Brock J. LaMeres, *The Modern Digital Design Flow*, pp. 1–12, Springer International Publishing, Cham, 2019.

[43] Ulrich Lauther, *Introduction to Synthesis*, pp. 1–14, Springer US, Boston, MA, 1992.

[44] Joseph C. Libby, Ashley Furrow, Paddy O'Brien, and Kenneth B. Kent, *A framework for verifying functional correctness in Odin II*, 2011 International Conference on Field-Programmable Technology, 2011, pp. 1–6.

[45] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz, *VTR 7.0: Next Generation Architecture and CAD System for FPGAs*, ACM Trans. Reconfigurable Technol. Syst. **7** (2014), no. 2.

[46] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Wei Mark Fang, and Jonathan Rose, *VPR 5.0: FPGA Cad and Architecture Exploration Tools with Single-Driver Routing, Heterogeneity and Process Scaling*, Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (New York, NY, USA), FPGA '09, Association for Computing Machinery, 2009, p. 133–142.

[47] Jason Luu, Conor McCullough, Sen Wang, Safeen Huda, Bo Yan, Charles Chiasson, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz, *On Hard Adders and Carry Chains in FPGAs*, 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines, 2014, pp. 52–59.

[48] A. Mishchenko, S. Chatterjee, and R. Brayton, *DAG-aware AIG rewriting: a fresh look at combinational logic synthesis*, 2006 43rd ACM/IEEE Design Automation Conference, 2006, pp. 532–535.

[49] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz, *VTR 8: High-Performance CAD and Customizable FPGA Architecture Modelling*, ACM Trans. Reconfigurable Technol. Syst. **13** (2020), no. 2.

[50] Konstantin Nasartschuk, Rainer Herpers, and Kenneth B. Kent, *Visualization support for FPGA architecture exploration*, 2012 23rd IEEE International Symposium on Rapid System Prototyping (RSP), 2012, pp. 128–134.

[51] V. Paruthi and A. Kuehlmann, *Equivalence checking combining a structural SAT-solver, BDDs, and simulation*, Proceedings 2000 International Conference on Computer Design, 2000, pp. 459–464.

[52] Maria Patrou, Jean-Philippe Legault, Aaron G. Graham, and Kenneth B. Kent, *Improving Digital Circuit Simulation with Batch-Parallel Logic Evaluation*, 2019 22nd Euromicro Conference on Digital System Design (DSD), 2019, pp. 144–151.

[53] Oleg Petelin and Vaughn Betz, *The speed of diversity: Exploring complex FPGA routing topologies for the global metal layer*, 2016 26th International Conference on Field Programmable Logic and Applications (FPL), 2016, pp. 1–10.

[54] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B. Kent, Peter Jamieson, and Jason Anderson, *The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing*, Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (New York, NY, USA), FPGA '12, Association for Computing Machinery, 2012, p. 77–86.

[55] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic, *Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs*, 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019, pp. 1–4.

[56] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead, *Designing Modular Hardware Accelerators in C with ROCCC 2.0*, 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, 2010, pp. 127–134.

[57] Sen Wang, *The improvement of the vtr project by using carry-chains and power specification*, Master's thesis, University of New Brunswick, 2013.

[58] C. Wolf, *Design and Implementation of the Yosys Open SYnthesis Suite*, Bachelor Thesis, Vienna University of Technology, 2013.

[59] Claire Wolf, *Yosys Open SYnthesis Suite*, `https://yosyshq.net/yosys/`.

[60] ———, *Yosys Manual*, `https://github.com/YosysHQ/yosys-manual-build/releases/download/manual/manual.pdf`, 2022.

[61] Claire Wolf, Johann Glaser, and Johannes Kepler, *Yosys-a free Verilog synthesis suite.*, Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip), 2013.

[62] Bo Yan, *High-level synthesis improvements and optimizations in odin ii*, Master's thesis, University of New Brunswick, 2014.

[63] Bo Yan and Kenneth B. Kent, *Hard block reduction and synthesis improvements in Odin II*, 2015 International Symposium on Rapid System Prototyping (RSP), 2015, pp. 126–132.

[64] Chi Wai Yu, Julien Lamoureux, Steven J.E. Wilton, Philip H.W. Leong, and Wayne Luk, *The Coarse-Grained / Fine-Grained Logic Interface in FPGAs with Embedded Floating-Point Arithmetic Units*, 2008 4th Southern Conference on Programmable Logic, 2008, pp. 63–68.

Appendix A

VTR Benchmarks

Table A.1 shows the detailed quality of results per each circuit for the VTR benchmarks.

circuit	VTR Flow Elapsed Time	Synthesis Time	Logic Depth	ABC Time	CLBs	Memories	Multipliers	Pre-Packed Blocks	Post-Packed Blocks	Device Grid Tiles	Pack Time	Place Time	Min Channel Width	Critical Path Routed Wire Length	Critical Path Delay
arm_core.v	0.76	3.45	1.11	0.40	0.84	1.04		0.78	0.88	0.85	1.35	0.78	0.98	0.87	1.27
bgm.v	0.71	4.27	1.00	0.26	0.99		1.00	0.98	0.99	1.00	1.16	1.02	1.00	1.00	1.03
blob_merge.v	0.45	5.16	1.00	0.06	0.80			0.68	0.84	0.82	1.07	0.71	1.13	0.90	1.01
boundtop.v	1.51	3.19	1.00	1.29	0.48			1.18	1.02	1.15	1.01	1.18	1.00	1.68	0.68
ch_intrinsics.v	0.60	0.31	1.00	0.21	1.05	1.00		0.96	1.01	1.00	0.70	0.85	1.00	0.77	0.69
diffeq1.v	0.55	0.63	0.83	1.24	0.97		1.00	1.00	1.00	1.00	0.74	0.92	0.93	0.98	1.00
diffeq2.v	0.50	0.57	0.83	1.08	1.05		0.71	0.98	0.99	0.79	0.90	0.80	1.09	0.74	0.96
mkDelayWorker32B.v	0.73	2.41	1.00	0.24	1.02	0.98		0.93	1.00	1.00	0.91	0.94	0.95	0.99	0.88
mkPktMerge.v	0.65	0.50	1.00	0.81	1.03	1.00		1.00	1.00	1.00	0.89	0.86	0.91	1.05	1.04
mkSMAdapter4B.v	0.75	2.11	1.25	0.83	0.93	1.00		0.97	0.98	1.00	1.02	1.08	1.12	0.93	1.17
or1200.v	0.65	1.21	1.00	0.66	0.96	1.00	1.00	0.95	0.96	0.93	0.68	0.87	0.98	0.98	1.03
raygentop.v	0.89	0.85	1.00	0.68	1.07		0.75	1.05	1.05	1.00	1.18	1.19	1.07	1.03	1.02
sha.v	0.05	1.17	1.33	0.01	0.85			0.91	0.90	0.89	0.83	0.92	0.94	0.80	1.08
spree.v	0.67	0.51	1.07	0.74	0.94	1.00	1.00	0.93	0.97	1.00	0.75	0.90	0.86	0.98	1.00
stereovision0.v	0.90	1.42	1.00	0.69	1.01			0.98	1.02	1.00	1.10	0.97	1.04	0.98	0.94
stereovision1.v	0.84	1.76	1.00	0.19	1.00		1.00	0.99	1.00	1.00	0.95	1.00	0.95	0.98	0.95
stereovision2.v	0.46	1.60	1.00	0.60	0.91		0.55	0.87	0.87	0.59	0.61	0.75	1.22	0.89	0.95
stereovision3.v	0.58	0.81	0.80	0.91	1.07			0.88	0.51	1.00	0.56	1.00	0.77	0.46	0.90
LU8PEEng.v	0.82	3.12	0.97	0.80	0.97	1.02	1.00	0.94	0.97	0.97	0.87	1.01	0.98	0.96	0.98
mcml.v	0.90	2.67	0.96	0.88	0.91	1.00	1.00	0.96	0.92	0.92	0.61	0.99	1.01	1.07	0.93
GEOMEAN	0.61	1.42	1.00	0.42	0.93	1.00	0.89	0.94	0.93	0.94	0.87	0.93	0.99	0.93	0.97
GEOMEAN(10k)	0.71	2.67	1.00	0.37	0.92	1.02	0.89	0.89	0.93	0.88	0.93	0.90	1.04	0.95	1.00

Table A.1: VTR benchmarks using Yosys as the elaborator and the Parmys plug-in as the partial mapper in the VTR flow. Results are relative to the baseline VTR setting.

Appendix B

Optimization in VTR Benchmarks

Table B.1 shows the QoR metrics for this experiment. In this table, the numbers are relative to the conventional VTR flow (Odin-II, ABC, VPR) with default settings. The first column in Table B.1 (labelled full optimization) is for the standard utilization of the Parmys plugin within the VTR flow where circuits are fully optimized before and after the Parmys pass. The second column (labelled pre-Parmys optimization) represents the experiment in which circuits are only optimized before the Parmys pass and no optimization is performed after the Parmys pass. The results in the third column (labelled post-Parmys optimization) are collected with optimization only after the Parmys pass and the metrics within the last column (labelled no optimization) are collected with no optimization in Yosys. The optimization in Odin-II space is being performed for all the columns and this provides the opportunity to amplify the importance and the efficiency of the Yosys optimizer.

	VTR			
	full optimization*	pre-Parmys optimization	post-Parmys optimization	no optimization
VTR Flow Elapsed Time	0.71	2.35	2.49	2.51
Synthesis Time	2.67	1.83	0.92	0.89
Logic Depth	1.00	1.00	1.00	1.03
ABC Time	0.37	0.53	0.70	0.70
CLBs	0.92	0.95	1.03	1.03
Memories	1.02	15.16	15.16	15.16
Multipliers	0.89	1.41	1.46	1.46
Pre-Packed Blocks	0.89	0.94	1.02	1.02
Post-Packed Blocks	0.93	1.13	1.20	1.20
Device Grid Tiles	0.88	2.96	3.11	3.11
Pack Time	0.93	1.08	1.09	1.07
Place Time	0.90	1.42	1.37	1.29
Route Time†	0.70	3.20	3.42	3.29
Min Channel Width	1.03	0.83	0.86	0.86
Wire Length	0.95	1.26	1.42	1.42
Critical Path Delay	1.00	1.04	1.06	1.06

Table B.1: VTR benchmarks using Yosys as the elaborator and the Parmys plug-in as the partial mapper in the VTR flow. Results are relative to the baseline VTR settings and the Geometric mean of the all circuits within each benchmark is reported.

* More than 3% change in the output is considered as a major change and highlighted in green if an improvement and highlighted in red if not.

† reported for the minimum channel width

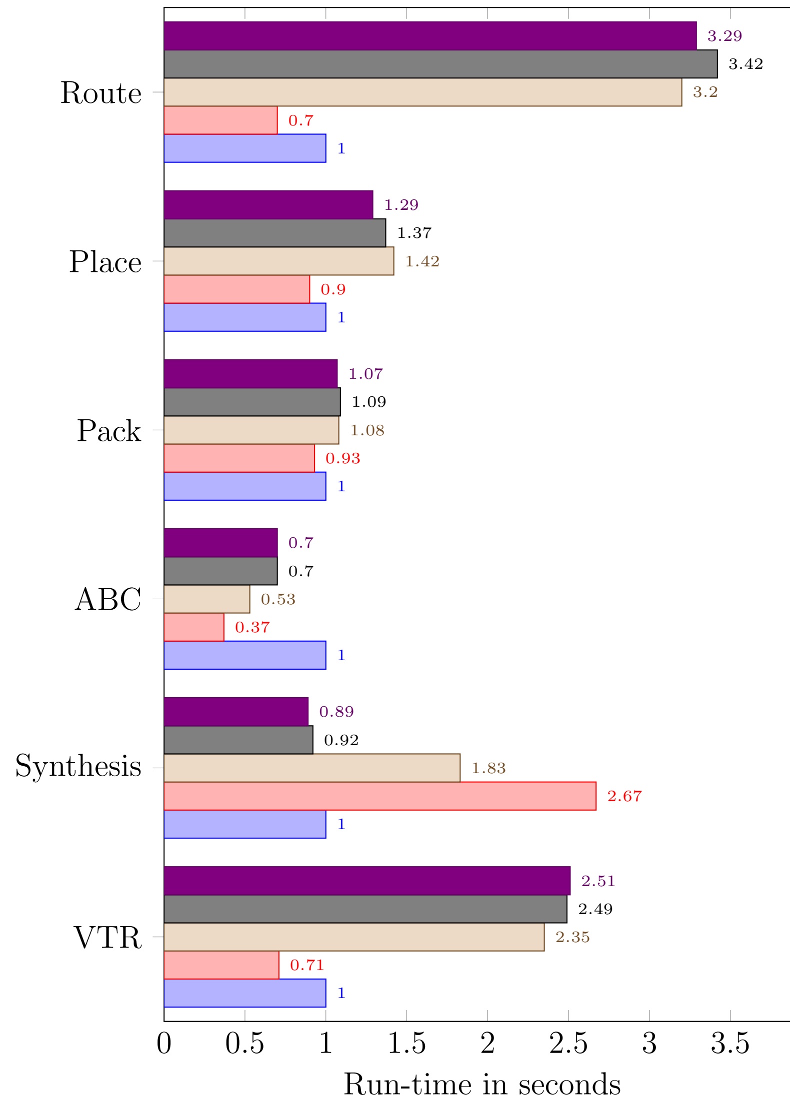

Figure B.1: Run-time comparison for VTR benchmarks utilizing the Odin-II frontend and the Parmys frontend with various optimization settings.

Appendix C

Koios Benchmarks

C.1 Default

Table C.1 shows the detailed quality of results per each circuit for the standard Koios benchmarks.

circuit	VTR Flow Elapsed Time	Synthesis Time	Logic Depth	ABC Time	CLBs	Memories	Multipliers	Pre-Packed Blocks	Post-Packed Blocks	Device Grid Tiles	Pack Time	Place Time	Min Channel Width	Critical Path Routed Wire Length
tpu_like.small.os.v	0.53	0.19	1.00	0.33	0.92	1.00		0.97	0.96	1.00	0.53	0.59		0.97
tpu_like.small.ws.v	0.57	0.15	0.71	0.30	0.67	1.00		0.82	0.82	1.00	0.92	0.50		1.02
dla_like.small.v	0.77	2.83	1.20	0.38	0.82	1.00		0.84	0.83	0.96	1.50	0.57		0.89
bnn.v	0.57	2.92	1.00	0.27	1.09			0.86	1.08	1.10	0.48	0.78		0.98
attention_layer.v	0.64	4.37	0.71	0.33	0.69	1.00		0.78	0.83	0.76	0.72	0.46		0.80
conv_layer_hls.v	0.42	0.25	1.00	0.28	1.01	1.00		1.00	1.00	0.96	0.27	0.81		0.94
conv_layer.v	0.41	1.56	1.00	0.27	0.86	1.00		0.92	0.89	1.00	0.55	0.51		0.95
eltwise_layer.v	0.53	2.64	1.00	0.12	0.99	1.00		1.02	0.99	1.00	0.60	0.67		0.95
robot_rl.v	0.25	0.04	0.83	0.17	0.68	0.88		0.81	0.76	1.00	0.50	0.44		0.76
reduction_layer.v	0.62	11.19	1.00	0.26	0.82	1.00		0.89	0.84	1.00	0.58	0.48		0.87
spmv.v	0.64	0.76	1.00	0.40	1.29	1.00		1.00	1.17	1.00	0.95	0.72		1.07
softmax.v	0.64	5.25	1.00	0.59	1.01			1.00	0.99	0.87	0.54	0.69		1.03
GEOMEAN	0.53	1.07	0.95	0.29	0.89	0.99	#NUM!	0.90	0.92	0.97	0.62	0.59	#NUM!	0.93

Table C.1: VTR benchmarks using Yosys as the elaborator and the Parmys plug-in as the partial mapper in the VTR flow. Results are relative to the baseline VTR setting.

C.2 No Custom Hard Blocks

Table C.2 shows the detailed quality of results per each circuit for the `no_hard_block` Koios task, which uses generic VTR hard blocks and no custom hard blocks.

circuit	VTR Flow Elapsed Time	Synthesis Time	Logic Depth	ABC Time	CLBs	Memories	Multipliers	Pre-Packed Blocks	Post-Packed Blocks	Device Grid Tiles	Pack Time	Place Time	Min Channel Width	Critical Path Routed Wire Length
tpu_like.small.os.v	0.50	4.79	0.80	0.33	0.95	1.00	0.96	0.96	0.97	1.00	0.62	0.73		1.02
tpu_like.small.ws.v	0.80	1.56	0.71	0.74	1.06	1.00	0.96	1.40	1.03	0.96	1.33	0.82		1.49
dla_like.small.v	0.42	2.45	1.20	0.36	0.81	0.60	0.88	0.82	0.81	0.82	0.52	0.54		0.87
bnn.v	0.58	2.68	1.00	0.27	1.08		1.00	0.86	1.07	1.09	0.43	0.71		1.06
attention_layer.v	0.51	4.09	0.71	0.13	0.58	0.87	0.72	0.68	0.74	0.82	0.46	0.43		0.77
conv_layer_hls.v	0.40	0.27	1.29	0.23	0.97	1.00	1.00	0.93	0.98	0.96	0.64	0.59		0.90
conv_layer.v	0.37	1.53	1.00	0.31	0.79	1.00	1.00	0.84	0.82	1.00	0.47	0.44		0.88
eltwise_layer.v	0.32	1.45	1.00	0.10	0.50	1.00	1.00	0.56	0.59	1.00	0.28	0.27		0.77
robot_rl.v	0.19	0.04	0.83	0.14	0.66	1.00	1.00	0.77	0.75	1.00	0.41	0.45		0.76
reduction_layer.v	0.59	9.90	0.83	0.20	0.79	1.00		0.75	0.80	1.00	0.75	0.43		0.89
spmv.v	0.77	0.70	1.00	0.32	0.78	1.17	1.00	0.71	0.88	1.17	0.49	0.46		1.04
softmax.v	0.61	5.28	1.00	0.39	0.95		1.00	0.91	0.96	0.96	0.67	0.63		1.02
GEOMEAN	0.47	1.54	0.93	0.26	0.81	0.95	0.95	0.83	0.86	0.98	0.55	0.52	#NUM!	0.94

Table C.2: Koios benchmarks targeting `k6_frac_N10_frac_chain_depop50_mem32K_40nm.xml` architecture using Yosys as the elaborator and the Parmys plug-in as the partial mapper in the VTR flow. Results are relative to the baseline VTR setting.

C.3 Multiple Architectures

Table C.3 shows the detailed quality of results per each circuit for the `multi_arch` Koios task, which utilizes 11 different targets for only the circuit named `conv_layer.v` to study the effect of the FPGA architecture on the QoR metrics.

architecture	VTR Flow Elapsed Time	Synthesis Time	Logic Depth	ABC Time	CLBs	Memories	Multipliers	Pre-Packed Blocks	Post-Packed Blocks	Device Grid Tiles	Pack Time	Place Time	Min Channel Width	Critical Path Routed Wire Length
k6FracN10LB_mem20K_complexDSP_customSB_22nm.xml	0.46	1.57	1.00	0.36	0.76	1.00		0.80	0.80	1.00	0.39	0.49		0.85
k6FracN10LB_mem20K_complexDSP_customSB_22nm.mem_heavy.xml	0.34	1.58	1.00	0.27	0.76	1.00		0.80	0.80	0.77	0.37	0.53		0.86
k6FracN10LB_mem20K_complexDSP_customSB_22nm.dsp_heavy.xml	0.44	1.64	1.00	0.34	0.76	1.00		0.80	0.80	0.73	0.25	0.94		0.84
k6FracN10LB_mem20K_complexDSP_customSB_22nm.densest.xml	0.33	1.50	1.00	0.26	0.76	1.00		0.80	0.80	0.77	0.26	0.47		0.76
k6FracN10LB_mem20K_complexDSP_customSB_22nm.denser.xml	0.34	1.56	1.00	0.25	0.76	1.00		0.80	0.80	0.95	0.39	0.56		0.82
k6FracN10LB_mem20K_complexDSP_customSB_22nm.coupled.xml	0.37	1.52	1.00	0.25	0.76	1.00		0.80	0.80	1.00	0.26	0.49		0.84
k6FracN10LB_mem20K_complexDSP_customSB_22nm.coupled.densest.xml	0.31	1.50	1.00	0.24	0.76	1.00		0.80	0.80	0.80	0.24	0.53		0.83
k6FracN10LB_mem20K_complexDSP_customSB_22nm.coupled.denser.xml	0.30	1.62	1.00	0.23	0.76	1.00		0.80	0.80	0.86	0.24	0.51		0.83
k6FracN10LB_mem20K_complexDSP_customSB_22nm.clustered.xml	0.33	1.70	1.00	0.24	0.76	1.00		0.80	0.80	1.00	0.24	0.50		0.87
k6FracN10LB_mem20K_complexDSP_customSB_22nm.clustered.densest.xml	0.31	1.87	1.00	0.23	0.76	1.00		0.80	0.80	0.84	0.26	0.53		0.80
k6FracN10LB_mem20K_complexDSP_customSB_22nm.clustered.denser.xml	0.35	1.80	1.00	0.27	0.76	1.00		0.80	0.80	1.00	0.23	0.53		0.84
GEOMEAN	0.35	1.62	1.00	0.26	0.76	1.00	#NUM!	0.80	0.80	0.88	0.28	0.54	#NUM!	0.83

Table C.3: The `conv_layer.v` circuit from the Koios benchmarks targeting 11 different architectures using Yosys as the elaborator and the Parmys plug-in as the partial mapper in the VTR flow. Results are relative to the baseline VTR setting.

Vita

Candidate's full name:
Daniel Khadivi

University attended (with dates and degrees obtained):
Bachelor of Computer Science–Software Engineering, University of Tehran, 2017
Master of Computer Science, University of New Brunswick, 2023