Odin II - An Open-source Verilog HDL Synthesis Tool for CAD Research

Peter Jamieson*, Kenneth B. Kent†, Farnaz Gharibian‡, and Lesley Shannon‡
*Dept. of Electrical and Computer Engineering
Miami University
Email: jamiespa@muohio.edu
†Dept. of Computer Science
University of New Brunswick
Email: ken@unb.ca
‡School of Engineering Science
Simon Fraser University
Email: fga7 or lshannon@ensc.sfu.ca

***Abstract*—In this work, we present Odin II, a framework for Verilog Hardware Description Language (HDL) synthesis that allows researchers to investigate approaches/improvements to different phases of HDL elaboration that have not been previously possible. Odin II's output can be fed into traditional back-end flows for both FPGAs and ASICs so that these improvements can be better quantified. Whereas the original Odin [1] provided an open source synthesis tool, Odin II's synthesis framework offers significant improvements such as a unified environment for both front-end parsing and netlist flattening. Odin II also interfaces directly with VPR [2], a common academic FPGA CAD flow, allowing an architectural description of a target FPGA as an input to enable identification and mapping of design features to custom features. Furthermore, Odin II can also read the netlists from downstream CAD stages into its netlist data-structure to facilitate analysis. Odin II can be used for a wide range of experiments; in this paper, we show three specific instances of how Odin II can be used by ASIC and FPGA researchers for more than basic synthesis. Odin II is open source and released under the MIT License.**

I. Introduction

Hardware description language (HDL) synthesis is an integral part of the computer aided design (CAD) flow used to map circuits to digital technologies such as Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs). HDL synthesis tools convert designs created in languages such as Verilog HDL [3] and VHDL [4] to a netlist that downstream stages of the CAD flow continue to map to the target technology.

We provide an HDL elaboration framework for synthesis research that is analogous to SUIF [5] for compiler research, VPR [6] for CAD research, and Simplescalar [7] for microprocessor architecture research; these types of environments are commonly used in the field of computing as they allow researchers to explore various design space options without having to build a complete tool every time. ODIN II allows researchers to investigate approaches/improvements to different phases of HDL elaboration that have not been possible previously.

In this work, we present the framework for Odin II, an open source front-end synthesis framework for mapping Verilog HDL designs that complements existing open source academic CAD flows. Odin II is built to integrate seamlessly with the VPR 5.0 [2] CAD flow, which allows researchers to explore FPGA architectures and CAD algorithms. However, Odin II is inherently a general synthesis framework that can be used to target any CAD flow. Odin II affords significant advantages over the original Odin [1] synthesis tool as Odin II provides a unified customizable front-end HDL synthesis environment. In particular, the original Odin used Icarus [8] as a front-end parser; for this reason, Odin was challenging to maintain and upgrade due to conflicting changes in the parser and back-end.

To demonstrate how this framework can be used for different research objectives, we include examples of possible experiments. Although it is possible to use Odin II for ASIC CAD flows, the remainder of this paper focuses on how Odin II can be used in FPGA CAD research using the following three experiments:

1) The front-end parser of Odin II is built using Bison [9] and Flex [10] tools. This parser provides additional hierarchical information, in the form of an abstract syntax tree (AST), beyond a flattened netlist. This AST can be used to help identify functional structures within in a design. This information can be used to map these structures to custom units and to gain a better understanding of the composition of larger and more complex designs. We illustrate this structural identification feature by showing how Odin II can identify finite state machines (FSMs) using the AST.
2) Odin II can read down flow CAD tool output files and store these netlists in the flexible netlist data-structure within Odin II. This feature can simplify CAD researchers work by providing a common netlist data-structure and a library of functions to manipulate this netlist. We demonstrate this feature by showing how a multiplier circuit can be described in a Verilog design, logically optimized by an external tool, read back in,

and manipulated by Odin downstream to estimate the number of transistors needed to implement each multiplier.

3) Odin II can read in a VPR 5.0 target technology library that describes the functional units available on the FPGA. These functional units need to be identified in the user's design and correctly mapped to the FPGA. In our example, we show how Odin II reads architecture files describing the availability and structure of hard multipliers on a target FPGA.

These experiments could not easily have been achieved without this type of framework (even with the original ODIN).

In addition to these features, Odin II is easily installed on NIX flavored systems and the tool allows us to leverage the availability of Verilog HDL benchmarks. All of these features and the design of Odin II allow researchers to use it in a number of innovative ways.

The remainder of the paper is organized as follows: In Section II we describe a typical CAD flow for FPGAs and some of the aspects of the FPGA architecture that are relevant to the CAD flow. In Section III, we describe how Odin II is designed. In section IV, we illustrate how the features of Odin II can be used in three specific cases, and finally, in Section V we conclude the paper.

II. Background

In this section, we describe the fundamental aspects of an academic FPGA CAD flow that Odin II is designed to target, and discuss existing front-end Verilog tools, open source projects, and how the Odin II framework is similar to other exploratory research frameworks.

A. FPGA CAD flow

Figure 1 shows an open source CAD flow used for VPR 5.0. A digital design is created in Verilog and used as the input to the flow. Odin II, the focus of this paper, parses the design and reads in the architecture file that describes the target technology to create a flattened netlist that consists of structures available for this architecture. Therefore, for the FPGA example, the netlist output from Odin II will consist of I/Os, logic gates, flip-flops, and hard circuits such as multipliers and memories. To map to structures such as multipliers, Odin II performs *inferencing* and *partial mapping* during which the tool identifies hard circuits in the Verilog design and then determines how to map them to the hard circuits on the FPGA architecture.

This netlist is then passed to the ABC tool [11] that first executes a logic optimization phase and then maps the logic into look-up tables (LUTs). The basic programmable cell of an FPGA is a Basic Logic Elements (BLEs), which itself is commonly a combination of a LUT and flip-flop [12].

The next stage of the CAD flow is called clustering and this stage packs LUTs and registers into clusters. Clusters are a collection of BLEs where the BLEs are connected to one another via intra-cluster routing. This intra-cluster routing is also connected to the inter-cluster routing that forms the remainder of the FPGA programmable routing architecture. T-Vpack [13] is the clustering tool that is used within the VPR 5.0 CAD flow.

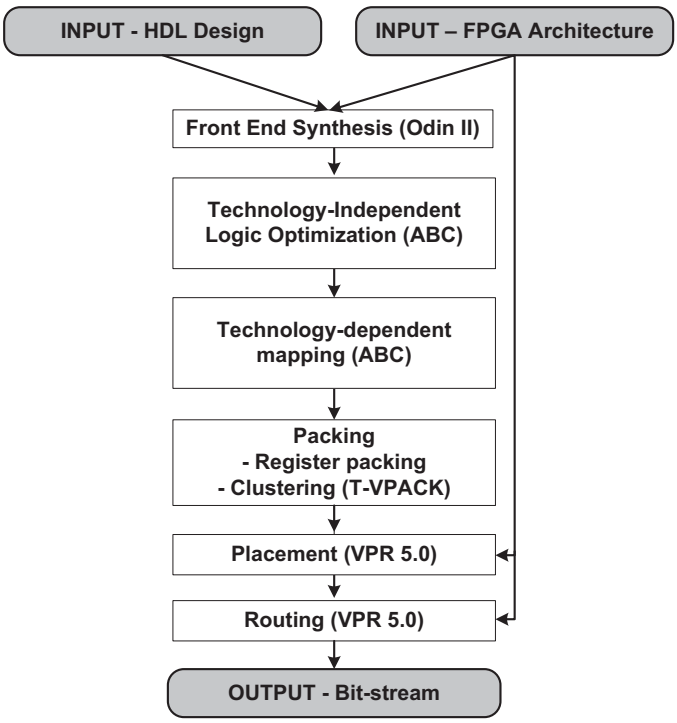

Fig. 1. A VPR 5.0 CAD flow

At this point the netlist consists of I/Os, clusters, and hard circuits, and these elements are placed onto the FPGA and routed using the VPR 5.0 tool, originally, created by Vaughn Betz [6]. The placer chooses the physical location of both the clusters and hard circuits on the Field-Programmable Gate Array (FPGA). Finally, the router chooses wire segments and activates the programmable switches as connection paths between placed elements. With the successful completion of placement and routing, the final output of the Computer Aided Design (CAD) flow is a configuration bit-stream that specifies the programming of the FPGA.

B. Verilog Synthesis and Academic Frameworks

Odin II is among a number of front-end HDL tools available for Verilog circuits. FPGA vendors including Altera [14] and Xilinx [15] and FPGA and ASIC CAD flow vendors including Magma [16], Synopsis [17], [18], and Mentor Graphics[19] all have front-end tools for both VHDL and Verilog in their commercial tool sets.

There are a number of available open source Verilog tools. A partial list includes Icarus [8], SAVANT [20], Veriwell [21], Verilator [22], and FreeHDL [23]. These tools are all open source, but they are not tools for synthesizing designs to a target technology, and instead, they mainly focus on the simulation and analysis of digital designs.

As described above, Odin II is a framework for Verilog HDL elaboration that can be quantified by passing its output

to a CAD flow that maps to a target technology. These types of exploratory frameworks such as SUIF [5] for compiler research, VPR [6] for FPGA architecture exploration, and Simplescalar [7] for computer architecture experimentation allow researchers to experimentally test out various ideas they have in a given area. Odin II has been created with this same objective: to provide an infrastructure that allows researchers to implement a range of experiments without having to create a new "tool". We describe some of these endeavors below, but first, we will describe the design of Odin II in more detail.

III. Odin II

A. Basic Tool Design

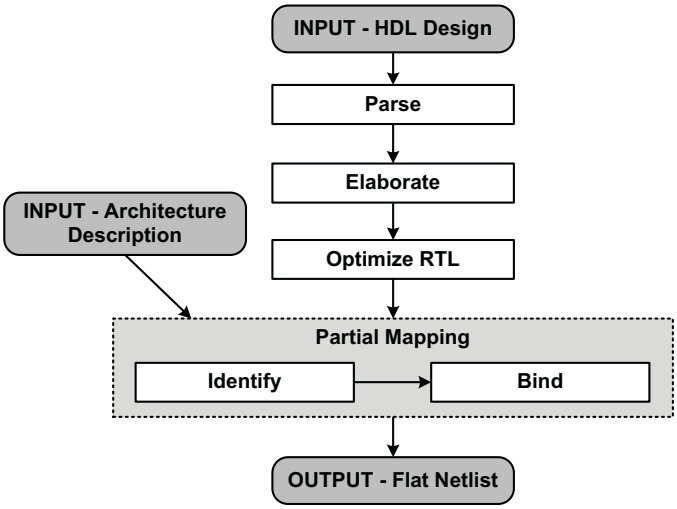

Fig. 2. The processing stages in Odin II

Figure 2 shows the major steps performed in Odin II to convert a Verilog HDL design into a flattened netlist. One of the main differences between Odin and Odin II is the inclusion of a dedicated parser in Odin II as opposed to using Icarus, a separately supported tool, as the parser in the original Odin. This change has significant impact on the ease-of-use, maintainability, and upgradeability of Odin II. For example, a simple feature such as keeping the design filename and line number associated with a logic gate can be maintained within Odin II; this was not possible in Odin and made simple features such as outputting syntax errors more difficult. It should be noted that Odin II was completely created from scratch and there is no usage of the original code in Odin in Odin II.

The parser in Odin II uses the Bison [9] and Flex [10] tools to convert the Verilog grammar description to a parser. Odin II uses the parser functions to build an abstract syntax tree (AST) [24] representing the entire inputted Verilog design. In Figure 2, this process of parsing and creating the AST is the parsing stage.

Figure 3 (b) shows the AST for the example Verilog code in Figure 3 (a). The figure shows how the Verilog is hierarchically stored in this data structure. What is not shown are the additional details that are stored for the line number and file that associates each AST node with the Verilog code.

With this parser, additional extensions to the Verilog HDL language can be more easily added and, more importantly, the later processing stages within Odin II have access to the hierarchical representation (the AST) of the design. We maintain this information throughout the elaboration process, meaning each node within a flattened netlist version of the design is directly associated with the AST node that it was derived from. Not only does this allow easier identification of structures such as FSMs (as we will describe later), but this information can be passed to later stages in the CAD flow so that designers can identify which lines of Verilog HDL designs impacted their designs.

Once the AST is created, an elaborator then traverses the AST to create a flattened netlist of the design (for more details on this process please see the original paper on Odin [1]). The netlist data-structure within Odin II that stores this netlist has been carefully designed to allow it to be easily manipulated. This is another key innovation in Odin II as opposed to Odin. In the process of HDL synthesis, optimizations cause the netlist to significantly change by a mixture of removal, additions, and rewiring of nodes. The netlist data-structure, therefore, needs to be designed to be easily manipulated. In Odin II, the netlist data-structure consists of unique objects for the hardware structures, the input and output pins connecting to each hardware structure, and the wires (or nets) inter-connecting pins. This design differs compared to the netlist data-structures in other CAD tools where the pin and wire information within the data-structure representing the hardware structure are embedded. This embedding approach compresses the amount of memory needed to store the netlist and speeds up processing the netlist due to locality, but it makes manipulations to the netlist much more difficult.

Figure 3 (c) shows a more detailed view of the flattened netlist for the example in Figure 3 (a) and the AST in Figure 3 (b), noting that the output pins for 'a' are not shown. A rounded rectangle is used to show hardware structures, an oval represents the pins, and a circle represents the nets. Using these three separate pieces of the netlist allows easy remapping of structures. For example, if one of the adders in Figure 3 (c) is replaced by some other structure then only the mapping from the original adder to the pin needs to be connected properly. If the pin and net were embedded in the adder, then a complex process of remapping all these entities would need to be done for the replaced node.

With both the flattened netlist and AST data structures representing the design, we can perform various optimizations on the design. Odin II does not yet support the wide range of compiler and logic optimizations that were available in Odin such as one-hot re-encoding of FSMs and arithmetic optimizations, but Odin II does do some basic compiler optimizations such as constant folding. See Figure I below for a comparison between optimizations in Odin and Odin II.

After compiler optimizations, Odin II performs partial-mapping, which determines how to pack design structures into

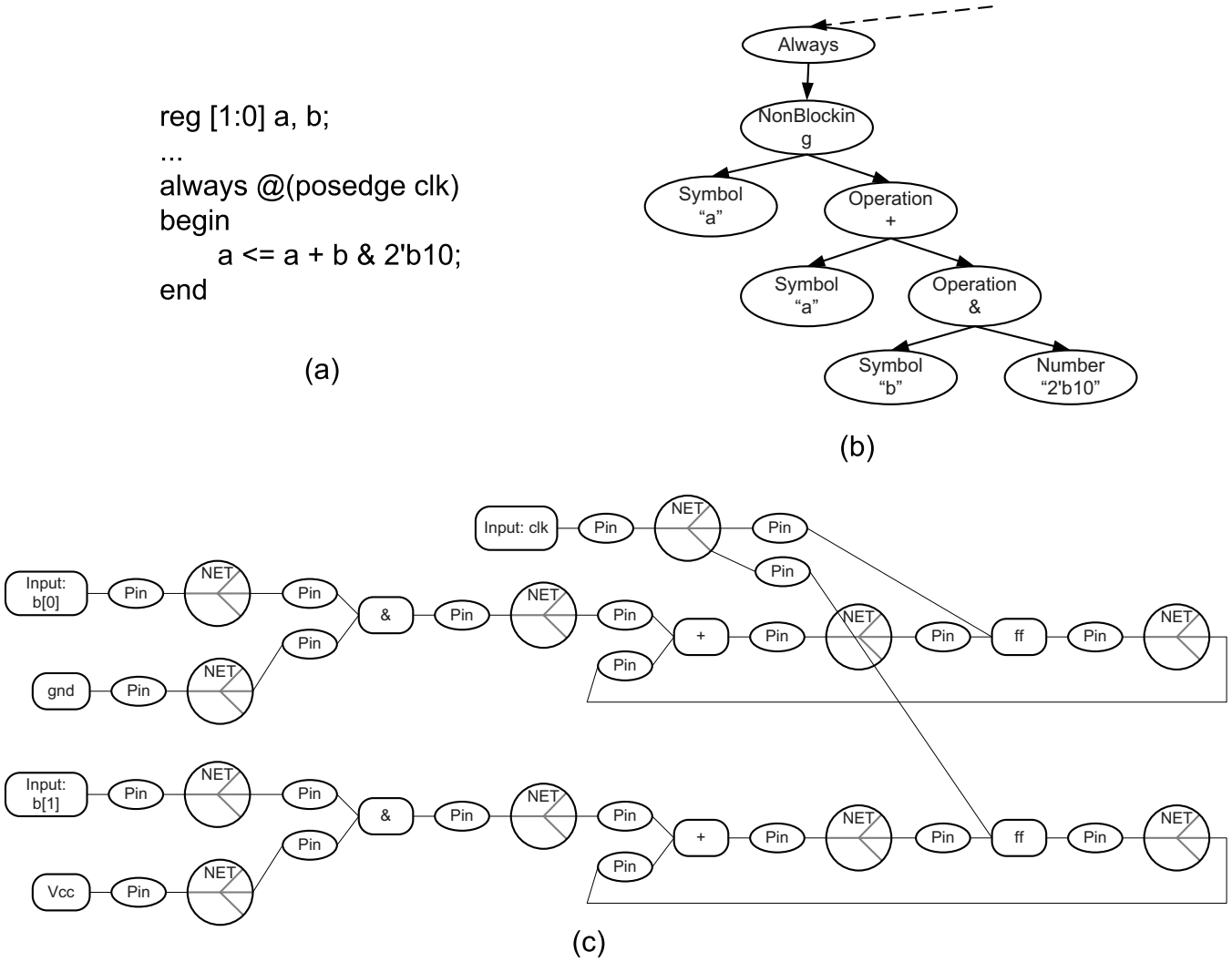

Fig. 3. The internal data structures in Odin II: (a) sample Verilog, (b) AST, (c) flattened netlist

the hard circuits available on the target technology (FPGA in our case). For example, if the target FPGA has 8 by 8 multipliers, then the partial mapper needs to find multipliers in the design and determine how to map them to 8 by 8 multipliers. This problem is even more challenging when there are a restricted number of hard circuits available on the target FPGA.

Partial-mapping does two things. First it extracts structures that are mappable to the hard circuits on the FPGA during what is known as the inferencing or identification stage. There are three common ways of doing this. The easiest method is to explicitly define how to instantiate hard circuits. In the case of a multiplier, the symbol "*" is used to instantiate a multiplier. Similarly, a library can be provided by the tool vendor such as the Library Parametrized Modules (LPMs) [25] used by Altera; the designer can then use these LPM structures to instantiate hard circuits. In the second method, designers are told to follow specified rules for writing Hardware Description Language (HDL) descriptions of complex circuits so that the tool can easily identify what circuits the designer intends to instantiate. For example, some synthesis tools specify how to write HDL statements so that flip-flops are identified by the partial mapper [18]. The final common method of identifying functions is an open-ended approach in which the synthesis tool uses matching techniques to extract complex circuits. This last method is the most difficult to implement and use due to both implementation complexity and run-time. Odin II can use all three methods, but does not currently have an implementation of the open-ended inferencing by sub-graph matching that is available in Odin.

Once the structures have been identified in the design, the next step of the partial mapper is to pack the structures into the hard circuits available on the FPGA. Odin II reads the architecture file used by VPR 5.0, which describes the hard circuits available on the FPGA. With this information, Odin II determines how to pack the structures for the target technology. For new hard structures, the user is responsible for building a packer for the proposed technology. For example,

adding floating point units to an FPGA fabric [26] would require that these structures be properly identified and packed in Odin II. Although floating point structures are not commonly found on FPGAs, and thus not supported as part of Odin II's basic framework, it is designed such that researchers are easily able to include these new structures for synthesis and mapping to their proposed target architecture.

Once all these stages are complete, then the final flattened netlist is output to a file. Odin II outputs designs in the Berkeley Logic Interchange Format (BLIF) format [27], but it is easy to output to other flattened netlist formats (internally in our tool) or using BLIF2vhdl or BLIF2verilog tools that will convert the BLIF output into a structural HDL format (consisting of only logic gates, wires, inputs and output), which is compatible with almost all CAD flows. The BLIF output file, in the FPGA CAD flow, is then read by ABC and the CAD flow continues to map the design to an FPGA as described in the previous section.

B. Comparison of Odin and Odin II

Table I shows a comparison between the synthesis tools, Odin and Odin II. The features and optimizations are listed in the first column and comments on whether Odin or Odin II has these features or supports these optimizations are listed in Columns 2 and 3 respectively. From the table, you can see that Odin II includes new features, but Odin II does not support all the original optimizations available in Odin. The reason for this is we built Odin II, completely, from scratch. The missing features in Odin II were implemented to bring Odin close to parity in terms of speed and area of mapped designs compared to the Quartus tool [1]. In the future, we will be adding these extensions to Odin II, noting that the software structure has been designed to easily add these features.

IV. Useful New Features in Odin II

In this section, we illustrate some of the new possibilities Odin II allows for with three experiments. These include identifying FSMs using the AST, reading downstream CAD outputs back into Odin II to estimate multiplier area, and identifying multipliers based on an FPGA architecture description file.

A. Finite State Machine Identification

As described earlier, identification of various hardware structures, part of partial-mapping, is important so that a synthesis tool can make the best decisions as to how to map these structures to the target technology. For example, FSMs when implemented on FPGAs with one-hot state encoding are both smaller and faster than most other state encoding schemes [28]. In this section, we show how Odin II identifies FSMs using the AST, and we explain how this is better than the approach taken in the original Odin.

The original Odin can identify FSMs and this identification is done by traversing the flattened netlist searching for feedback paths that satisfied specific rules. The problem with this approach is that it takes significant processing time to do the search and the algorithmic implementation is complex, allowing for the possibility that some FSMs might be missed. A faster way of identifying FSMs is to use the AST and associated symbol table. Specifically, by examining always blocks and case statements in the design and using the symbol table to identify where and how candidate registers are used in the design, we can analyze candidate state registers to find FSMs.

The basic approach to search for an FSM in the Odin tools is to identify candidate state registers and determine if there is a combinational feedback path from the candidate register to itself. Note, both Odin and Odin II search for FSMs based on similar principles, but the feedback search is simplified when using the AST and the symbol table. The same search using the flattened netlist requires a depth first search that terminates only when all combinational paths have been searched.

To illustrate the quality of our method, we pass ten benchmarks through the Odin II flow and Altera's synthesis tool, Quartus II, version 9.0 web edition [29]. Both tools identify the number of FSMs that are in a design.

Table II shows the number of FSMs identified by Quartus and Odin II. Column 1 shows the benchmark name: benchmarks "glue" and "io_expander" are from the GroundHog 2009 benchmark suite [30], benchmark "iir" is from OpenCores [31], the raytrace benchmarks [32] and stereo_vision [33] were created at the University of Toronto. Column 2 shows the number of *unique* FSMs in the benchmark. Column 3 and 4 show the number of FSMs identified by Quartus and Odin II respectively, where Quartus identifies the number of instantiated FSMs and Odin II identifies the number of unique FSMs. Note that Column 2 and Column 4 match, meaning Odin II identifies FSMs with 100% accuracy, and the value differences between column 3 and 4 are described below.

In general, the identification methodology in Odin II identifies all FSMs as we can see in the second and fourth column match. In some cases, it appears that Quartus identifies more FSMs in the design than Odin II. The reason for this difference is based on how the two tools report identified FSMs. Quartus describes the number of instantiated FSMs where as Odin II describes the number of unique FSMs in the design. For example, if a module has an FSM in it and is instantiated twice by a higher level module, then Quartus would identify 2 FSMs and Odin II would identify 1 FSM. Both are correct in terms of their respective context and it is a simple matter of the generated reports.

B. Reading Output of Other CAD Stages

Odin II can read in the output files from downstream CAD tools into the netlist data-structure within Odin II. For example, Odin II can read in the outputs from the academic FPGA CAD flow after tech-mapping (the output of ABC) and clustering (the output of T-Vpack). These outputs are read into the same flattened netlist data-structure, described earlier. This feature is included in Odin II to facilitate a number of

TABLE I
FEATURE LIST FOR ODIN AND ODIN II

Feature or Optimization	Odin	Odin II
Target CAD flows	VPR 5.0 and Quartus	VPR 5.0
Reads Architecture File	No	**Yes**
Internal Parser	External (Icarus)	**Yes**
Flexible Netlist Data Structure	No	**Yes**
Read Later CAD flow output	No	**Yes**
Multiplier Packing	**Yes**	**Yes**
Memory support	No	In Development
Dead node warnings	No	**Yes**
Multiplexer Motion/Collapsing	**Yes**	No
FSM identification	**Yes**	**Yes**
FSM one-hot encoding	**Yes**	No
Sub-graph Matching	**Yes**	No
Algebraic Simplification	**Some for add and multiply**	No
Constant Folding	No	**Yes**
Common Sub-expression Elimination	No	No
Strength Reduction	No	No

TABLE II
FSM IDENTIFICATION COMPARISON

Benchmark	# of unique FSMs	# of FSMs Altera	# of FSMs Odin II
glue	3	3	3
iir	1	1	1
io_expander	5	5	5
raytrace_0	7	9	7
raytrace_1	3	5	3
raytrace_2	7	7	7
raytrace_3	4	5	4
stereo_vision_0	0	0	0
stereo_vision_1	2	48	2
stereo_vision_2	0	0	0
stereo_vision_3	1	1	1

possibilities, and the stages of the FPGA CAD flow that can, currently, be read into Odin II are shown in Figure 4.

To illustrate how this feature might be useful, imagine that you want to compare different multiplier designs in terms of area in minimum width transistor counts. In this case we want to compare 36x36, 18x18, and 9x9 with and without booth encoding [34]. You could achieve this in a number of ways including finding standard cell implementations of the multipliers and their associated area or building an equation to estimate the number of transistors for each of the 6 multipliers.

For our approach, we use Odin II to find the transistor estimates of the six multipliers in the following way. First, we create a Verilog HDL design that includes the statement "a = b * c;". By changing the size of the input and output pins (b, c, and a) we can change the size of the instantiated multiplier. We input these 3 designs into Odin II and configure Odin II to map multipliers into logic gates using either booth encoding or not. The output from Odin II for each of these six designs is a BLIF file containing a gate level implementation of a 9x9, 18x18, and 36x36 multiplier with and without booth encoding. Next, we pass these 3 designs through ABC to perform logic optimization. The logic optimized designs are read back into Odin II into the flattened netlist data-structure.

We built a custom function in Odin II that counts the number of minimum width transistors needed to implement a design that is purely implemented with logic gates. For simplicity, we assume a CMOS implementation of all logic gates and that the P-doped transistor are 2 minimum width transistors.

Table III shows the minimum width transistor counts for each of the six multipliers passed through the Odin II flow and calculated based on an equation. The key difference between column 3 (Odin II flow) and column 4 (equation based) is that the equation based approach overestimates the number of transistors in a multiplier. We believe the reason for this overestimation is that the logic optimization tool performs area optimizations that reduce the number of gates and the resulting number of transistors.

This experiment illustrates how reading in outputs from downstream CAD flow can be used to quickly and more accurately estimate area. Another idea we have for this feature is Odin II could read downstream outputs and make connections between the original Verilog design and later downstream outputs. These connections will help us understand what the algorithms in the CAD flow have done. For example, a module in the initial Verilog design could be identified after clustering by showing which FPGA clusters contain pieces that implement the module. This would help designers understand how their designs are being optimized by each stage of the CAD flow.

Finally, we believe that the most important use of this

TABLE III
TRANSISTOR ESTIMATES FOR 9X9, 18X18, AND 36X36 MULTIPLIERS

Multiplier Size	Encoding	Minimum Width Transistors Odin II	Minimum Width Transistors Equation
9x9	Normal	4592	4617
9x9	Booth	1702	1769
18x18	Normal	17932	18468
18x18	Booth	6331	6453
36x36	Normal	72203	73782
36x36	Booth	23770	24570

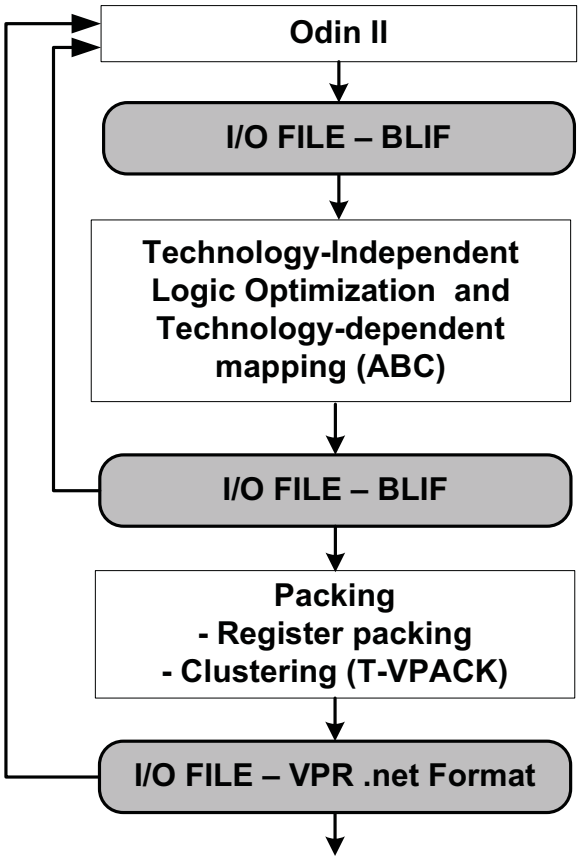

Fig. 4. The outputs in the CAD flow Odin II can read

feature will allow researchers to implement post-processing CAD algorithms using Odin II. Though researchers can take time to learn each of the CAD flow tools in an open source academic flow and implement their algorithms, in some cases, a simple algorithm might need to be quickly implemented (this is similar to scripting versus programming). Odin II can speed up implementing these circuit "scripts" since researchers can use the common netlist data-structure and functions to manipulate this netlist in Odin II. For example, we used this feature as part of the power estimation framework (to do the activation estimation) in an updated VPR 5.0 with power estimation (http://www.users.muohio.edu/jamiespa/vpr_5_pow.html).

C. Using Target Technology Architecture Description

As described earlier, one of the key steps in a CAD flow is identifying and mapping functional units in a design to the target technology. There is no clear conclusion on which stage of the CAD flow should do this mapping, but HDL synthesis contains sufficient high-level information of the design to perform this action. In Odin, partial-mapping was done by creating a proprietary library that described the hard circuits available on a target FPGA. This approach, however, required that researchers create both this library and FPGA architecture description file to match exactly. In Odin II, we have added the feature so that Odin II directly reads VPR 5.0 FPGA architecture files, and uses the information in this file to identify and map hard circuits.

To illustrate this feature, we used Odin II to identify multipliers for an FPGA with 9x9 hard multipliers and an FPGA with hard 18x18 multipliers in the ten benchmarks used in the FSM experiment above. We created two architectures files for VPR 5.0 that describe an FPGA with a 9x9 multiplier and an FPGA with 18x18 multiplier, and we input these architecture files along with the benchmarks into Odin II. Odin II identifies multipliers based on the "*" symbol, and then uses a packing algorithm to determine how to map the different sized multipliers in the design to the multipliers available on the FPGA. For example, a 10x10 multiplier in the design will be mapped to one 9x9 hard multiplier and some soft logic on an FPGA instead of two 9x9 multipliers, because the previous approach is both faster and consumes less resources.

TABLE IV
NUMBER OF HARD MULTIPLIERS USED WHEN PACKED BY ODIN II

Benchmark	# of hard 9x9	# hard 18x18
glue	2	1
iir	14	5
io_expander	0	0
raytrace_0	0	0
raytrace_1	0	0
raytrace_2	45	18
raytrace_3	0	0
stereo_vision_0	0	0
stereo_vision_1	152	152
stereo_vision_2	852	564
stereo_vision_3	0	0

Table IV shows multiplier packing results for each of the 10 benchmarks run through Odin II. Column one shows the benchmark name, and columns two and three shows the

number of hard 9x9 multipliers and 18x18 multipliers used by the design when mapped to both types of FPGAs.

For each of these benchmarks we can see that, depending on the type of hard multiplier available on the FPGA there is a varying number of multipliers used. Note that it takes 4 hard 9x9 multipliers to implement an 18x18 multiplier, and in each of the benchmarks that uses multipliers there are less than four times as many used 9x9 hard multipliers than used 18x18 multipliers. The reason for this is that each benchmark contains varying sizes of multipliers, and only in the case when a benchmark contains only 18x18 multipliers (or larger) will we see the number of hard 9x9 multipliers be exactly four times the number of 18x18 multipliers.

The benchmark stereo_vision_1 shows that there are an equal number of used 9x9 and 18x18 hard multipliers. The reason for this is the multipliers in this benchmark are all sized less than or equal to a 9x9 multipliers. In the FPGA with 18x18 hard multipliers it is more efficient to implement these design multipliers on an 18x18 multiplier if available.

V. Conclusion

ODIN II provides researchers with a much needed framework for HDL elaboration tool investigation. Although the original Odin provided an initial open source tool that achieved results similar to commercial tools, it was inflexible and, therefore, not suited to CAD research.

Odin II is an open source HDL elaboration environment that converts Verilog HDL designs and maps them to CAD flows targeting ASICs and FPGAs (currently targets VPR 5.0 FPGA exploration). A significant effort has been made to make this tool useful to researchers and designers. Odin II is a significant improvement over its predecessor Odin, and this was achieved by including a parser and focusing on the design of the software tool, including the netlist data-structure.

Odin II also includes features that will allow researchers to learn and implement new ideas for mapping designs to FPGAs. We illustrated three of these features including experimental results.

Odin II source code, regression benchmarks, and more documentation can be found at http://www.users.muohio.edu/jamiespa/odin_II.html.

References

[1] P. Jamieson and J. Rose, "A Verilog RTL Synthesis Tool for Heterogeneous FPGAs," in *Field-Programmable Logic and Applications*, 2005, pp. 305–310.

[2] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, and J. Rose, "VPR 5.0: FPGA CAD and Architecture xploration Tools with Single-Driver Routing, Heterogeneity and Process Scaling," in *ACM/SIGDA International Symposium on FPGAs*, Feb 2009.

[3] *Verilog Hardware Description Reference*, Open Verilog International, March 1993.

[4] *IEEE Standard VHDL Language Reference Manual*, IEEE, 1987.

[5] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. wei Liao, C. wen Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," *ACM SIGPLAN Notices*, vol. 29, pp. 31–37, 1994.

[6] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[7] C. I. Simplescalar, D. Burger, and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," Tech. Rep., 1997.

[8] S. Williams, "ICARUS Verilog at http://www.icarus.com/eda/verilog/," 2007.

[9] GNU, "Bison - GNU parser generator," http://www.gnu.org/software/bison/, 2009.

[10] Vern Paxson, "The Lex & Yacc Page," http://dinosaur.compilertools.net/flex/index.html, 2009.

[11] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Transactions on CAD*, vol. 26, no. 2, pp. 240–253, 2007.

[12] A. Singh and M. Marek-Sadowska, "Efficient Circuit Clustering for Area and Power Reduction in FPGAs," in *ACM/SIGDA International Symposium on FPGAs*, 2002, pp. 59–66.

[13] A. Marquardt, V. Betz, and J. Rose, "Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density," in *ACM/SIGDA International Symposium on FPGAs*, Monterey, CA, 1999, pp. 37–46.

[14] Altera Corporation. 101 Innovation Drive, San Jose CA 95134, "1996 Data Book," 1996.

[15] "XILINX at http://www.xilinx.com."

[16] Magma Design Automation Inc., "Blast FPGA," 2005.

[17] Synopsys, "Design Compiler FPGA," 2004.

[18] Synplicity, "Synplify Pro," 2003.

[19] Mentor Graphics, "LeanardoSpectrum," 2001.

[20] SAVANT, "SAVANT: VHDL Analysis Tools," http://www.ece.uc.edu/~paw/savant/, 2009.

[21] Veriwell, "Verilog Simulator," http://sourceforge.net/projects/veriwell/, 2009.

[22] Wilson Snyder and Duane Galbi and Paul Wasson, "Introduction to Verilator," http://www.veripool.org/wiki/verilator, 2009.

[23] FreeHDL, "A project to develop a free, open source, GPL'ed VHDL simulator for Linux!" http://www.freehdl.seul.org/, 2009.

[24] J. Jones, "Abstract syntax tree implementation idioms," in *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)*, 2003.

[25] "Electronic Industries Association standard for Library Parametrized Modules," 1993, http://www.edif.org/lpmweb/intro/what_is_lpm.htm.

[26] C. W. Yu, A. Smith, W. Luk, P. Leong, and S. Wilton, "Optimizing coarse-grained units in floating point hybrid fpga," in *ICECE Technology, 2008. FPT 2008. International Conference on*, Dec 2008, pp. 57–64.

[27] U. of California Berkeley, "Berkeley Logic Interchange Format (BLIF)," 1992.

[28] S. Golson, "One-hot state machine design for FPGAs," in *3rd PLD Design Conference*, Santa Clara, CA, Mar. 1993, pp. 1–6.

[29] Altera, *Quartus II Handbook, Volumes 1, 2, and 3*, 2004.

[30] P. Jamieson, T. Becker, W. Luk, P. Cheung, and T. Rissa, "Benchmarking Reconfigurable Architectures in the Mobile Domain," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2009.

[31] "http://www.opencores.org," 2007.

[32] J. Fender and J. Rose, "A High-Speed Ray TRacing Engine Built on a Field-Programmable System," in *IEEE International Conf. On Field-Programmable Technology*, 2003, pp. 188–195.

[33] A. Dharabiha, J. Rose, and W. MacLean, "Video-Rate Stereo Depth Measurement on Programmable Hardware," in *IEEE Computer Society Conference on Computer Vision & Pattern Recognition*, 2003, pp. 203–210.

[34] A. D. Booth, "A signed binary multiplication technique," *Quarterly Journal of Mechanics and Applied Mathematics*, no. 2, pp. 236–240, 1951.