# An Open Source Circuit Library with Benchmarking Facilities

Mariusz Grad and Christian Plessl
Paderborn Center for Parallel Computing, University of Paderborn
{mariusz.grad|christian.plessl}@uni-paderborn.de

***Abstract*—In this paper, we introduce the open-source PivPav backend tool for reconfigurable computing. Essentially, PivPav provides an interface to a library of digital circuits that are kept in a database. For each circuit, an extensive set of metadata is made available through an application programming interface. PivPav is agnostic of the kind of circuits that are stored in the library, but it provides an interface to commonly used circuit generators. The integrated benchmarking framework allows for processing and measuring the circuits with FPGA CAD tools under a variety of implementation constraints. This process characterizes each circuit, for example with its latency, maximum operating frequency, power consumption, input and output interfaces, etc. While PivPav is useful as a stand-alone benchmarking tool, its main purpose is to act as a building block in higher-level tool flows for reconfigurable computing. We present three case studies, that illustrate the use of PivPav for the purpose of instruction-set extension, data path generation, and design space exploration.**

## I. Introduction

In this paper, we introduce the PivPav tool for reconfigurable computing. The main purpose of PivPav is to provide a software infrastructure for storing and retrieving circuits and meta information about the circuits, for example, speed, latency, and resource usage, in a database. To this end, PivPav combines three components, which are illustrated in Figure 1.

- The **circuit factory** is responsible for generating circuits. For this purpose, PivPav interfaces to the widely used circuit generators Xilinx Coregen [1] and FloPoCo [2] and supports importing circuits that have been generated by synthesis from hardware description languages.
- The **benchmarking framework** is responsible for determining the performance and metadata for the circuits obtained from the circuit factory by executing the FPGA CAD tool flow and analysis tools for gathering characteristic metrics of each circuit. Overall, more than 90 different metrics are determined for each circuit, for example, the resources used by the circuit, the maximum frequency, or the power consumption. Additionally, the information about the optimization settings of the FPGA tool flow (e. g., optimization goals and effort) and the runtime and memory usage of the design tools are determined.
- The **circuit library** is the database component that stores the circuit and the results of the benchmarking process, including the circuit's source code, netlist, parsed VHDL top entity, results files from the FPGA CAD tool flow, and the CAD tool flow characterization data.

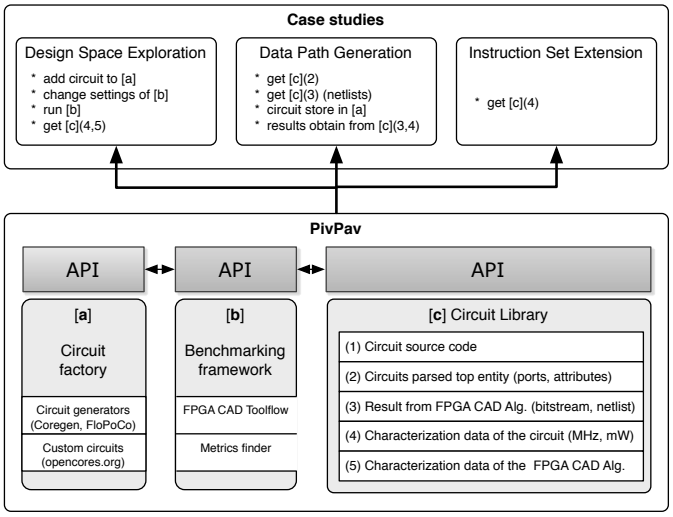


Figure 1: Overview of the PivPav tool and outline how the case studies are using the tool

While PivPav is useful as a standalone tool for benchmarking circuits, it is intended to be used as a backend component in a tool flow for the purpose of selecting a specific circuit from a large library based on functional and non-functional preferences. A prime example of such an application is data path synthesis, which was also our use case that triggered the development of PivPav. In data path synthesis, complex data paths are assembled from circuits representing basic operators, where each operator can be available in a large variety of implementations that are functionally equivalent, but differ in hardware size, speed, latency, etc. For finding the optimal data path implementation, a design space exploration of the basic operators is required, which is well supported by PivPav.

For integrating PivPav in custom design tools, PivPav features an application programming interface (API) that allows for accessing the circuit library from C and C++ programs. We release our tool under the GPL open source license allowing the community to freely benchmark and test their designs and FPGA CAD algorithms and to build higher level design tools that leverage the functionality of PivPav.

Published in: *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*

## II. Related tools

Circuit libraries play an important role for increasing the design productivity by allowing designers and design tools to reuse existing circuits as pre-generated components. Libraries providing different implementation alternatives further facilitate selecting the optimal circuit from a choice of functionally equivalent circuits, which provide different performance vs. area vs. latency trade-offs. While the circuit libraries used by HDL synthesis tools are typically not directly accessible by the designer, the FPGA tool vendors offer standalone circuit generators, such as Xilinx Coregen [1] or Altera MegaWizard [3] which generate circuits tailored for the respective FPGA architectures. Other circuit generators, such as FloPoCo [2] or GRLIB [4], support multiple FPGA families or generate architecture independent circuits.

PivPav differs from these tools by integrating a programmable circuit library, benchmarking infrastructure, and a circuit factory in a single tool. It allows retrieving a large set of meta data about each circuit, such as performance, area, power consumption, latency, etc., which provides the necessary input data to tools that estimated the performance of a larger circuit that is composed of elements retrieve from PivPav. Additionally, PivPav allows the user to programmatically connect individual circuits to form larger data paths and to generate the corresponding, structural HDL code. While all performance-driven high-level synthesis tool are likely to use a similar library internally, to the best of our knowledge PivPav is the only open-source tool that offers this functionality.

## III. Case Studies & Results

In this section, we evaluate PivPav with three case studies that demonstrate how we use PivPav in our ongoing research projects. The case studies cover three areas: instruction-set extension (III-A), data path generation (III-B), and design space exploration (III-C). As presented in Figure 1, each case study has different requirements and thus exercises different parts of the PivPav tool. However, all of these requirements can be met by a single tool.

### A. Instruction-Set Extension

In our previous work, we have studied a software and hardware tool flow for a dynamically reconfigurable instruction-set extension architecture named *Woolcano* [5]. A limitation of our proposed tool flow was the absence of an automated method for profiling the application and selecting the optimal custom instructions. This problem is known as the instruction-set extension problem and was performed manually in our previous work. To eliminate this limitation we have developed a tool which can identify and extract the most suitable functions to be implemented as custom instructions on the Woolcano architecture [6].

One of the most interesting features of the developed tool is the possibility to estimate the performance when a given part of the software is translated to an application-specific custom instruction. To this end, the tool estimates the time for executing the function of the custom instruction in software and hardware.

The most challenging obstacle in the estimation process is computing an accurate performance estimate, in terms of latency and maximum operating frequency, for the generated hardware circuit that implements the custom instruction. Since the generated hardware implements the same data flow graph as the software function, our tool flow follows the approach of translating each PowerPC instruction to an equivalent *hardware operator* and composing a data flow graph from these operators. The performance of the resulting custom instruction can be estimated by computing the critical path in this data flow graph, or for pipelined operators, by finding the operator with the minimum operating frequency. The timing overhead introduced by connecting two operators via the FPGA's routing fabric is modeled as a technology specific constant that is determined experimentally.

Thus for estimating the hardware performance we need performance metrics for each basic *hardware operator* circuit, which is exactly a use case for the PivPav tool. For some selected operators, for example, floating point operators from IP Core Libraries, data sheets that characterize each operator with the required performance metrics are available, e. g., [7]. For most other operators, such as integer, logic, or bit manipulation operations, no data sheets are available. This is particularly the case for operators that are created on-demand by HDL synthesis tools. Moreover, the characterization data in data sheets are not exhaustive and do not cover all FPGA families, models, and speed grades. But even within one device family, the performance of operators can vary significantly. For example, the data sheet for Xilinx Coregen quotes the maximum speed of the floating-point multiplier as 192 MHz, or 423 MHz respectively, for two FPGA models from the Xilinx Virtex-6 family [7, Table 23 vs Table 25]. This huge range makes it impractical to estimate accurate performance metrics for devices that are not even tabulated in the data sheets.

The PivPav tool addresses this problem by generating a library of operators, where each operator is specifically optimized for the targeted FPGA family. The performance and many other metrics are automatically extracted from the implementation. These metrics are made available to the instruction-set extension tool via the PivPav API, which is illustrated in Figure 1. Since all data are generated beforehand with the benchmarking facilities of the tool, there is no need for the instruction-set extension tool to run the FPGA CAD tool flow in the estimation process. In Table I, we present an excerpt of the result when querying the PivPav library for the Xilinx XC4VFX100-FF1152-10 device.

*Conclusion:* PivPav is able to deliver accurate metrics from different domains concerning the performance, power consumption or circuit area for every circuit stored in the library. A part of these metrics is needed by tools which would like to reliably and accurately model the hardware. PivPav can be treated in that role as a data source and can

| Circuit | Latency | Initiation interval | Pwr . cons. | Max FRQ after PAR |
|---|---|---|---|---|
| | [ cyc ] | [ cyc. ] | [ mW ] | [ MHz ] |
| add_561 | 4 | 0 | 1244.7 | 169.3 |
| add_558 | 1 | 0 | 1242.8 | 107.4 |
| mul_376 | 5 | 0 | 1259.2 | 190.7 |
| mul_403 | 1 | 0 | 1260.1 | 43.4 |
| sub_19 | 18 | 0 | 1246.4 | 179.1 |
| sub_1 | 1 | 0 | 1237.0 | 104.5 |
| div_104 | 50 | 1 | 1344.6 | 98.8 |
| div_158 | 43 | 2 | 1321.6 | 41.0 |
| fpadd_1183 | 7 | 0 | 1264.0 | 139.0 |
| fpadd_1161 | 1 | 0 | 1296.3 | 31.1 |
| fpmul_1115 | 10 | 0 | 1316.2 | 136.8 |
| fpmul_1136 | 1 | 0 | 1289.8 | 40.3 |
| fpsqrt_1011 | 26 | 1 | 1072.9 | 175.5 |
| fpsqrt_691 | 3 | 1 | 1074.0 | 29.7 |
| fpsub_661 | 16 | 0 | 1295.2 | 139.4 |
| fpsub_675 | 1 | 0 | 1278.2 | 30.5 |
| fpdiv_1595 | 28 | 19 | 1254.6 | 171.9 |
| fpdiv_1209 | 4 | 4 | 1267.1 | 32.8 |
| fp_float2int_630 | 4 | 0 | 1062.7 | 172.0 |
| fp_float2int_627 | 1 | 0 | 1070.3 | 43.3 |

Table I: Part of metrics requested by instruction-set extension tool for the XC4VFX100-FF1152-10 FPGA.

be used as a backend to these tools. Since the library already comprises the circuits which are equivalent to PowerPC and Microblaze instructions the tool matches the requirements of the instruction-set extension domain very well.

### *B. Data Path Generation*

A data path generator assembles complex data paths from circuits representing basic operators. The basic operators are stored in the PivPav library and each operator can exist in a variety of implementations that are functionally equivalent but differ in hardware resources, speed, latency, etc.

The data path generator accesses the PivPav circuit library and creates a structural HDL description that instantiates and wires the appropriate operators. While this may look like a straightforward task at the surface, connecting a wide variety of hardware operators from different sources bears a number of practical difficulties since the interface of the components differs in data types and widths, naming of IO ports, polarity of reset signals for sequential operators, differences in latency, etc. This difference can be seen between the behavioral (a) and structural (b) codes presented in Figure 2.

The results of these implementations are shown in Table II. The first implementation variant, denoted as Behavioral, is generated by synthesizing the behavioral description from Figure 2a with the Xilinx XST VHDL synthesis tool. This implementation is purely combinational and uses the operators inferred by the synthesis tool instead of custom operators from the PivPav library. Additionally, we generate two pipelined implementations of the data path, denoted as Pipelined(1) and Pipelined(2), from the structural VHDL description in Figure 2b. These implementations use circuits 558 and 403 for Pipelined(1), and circuits 561 and 376 for Pipelined(2). The characteristics of these basic hardware operators are shown in Table I. It should be emphasized here, that the generator does

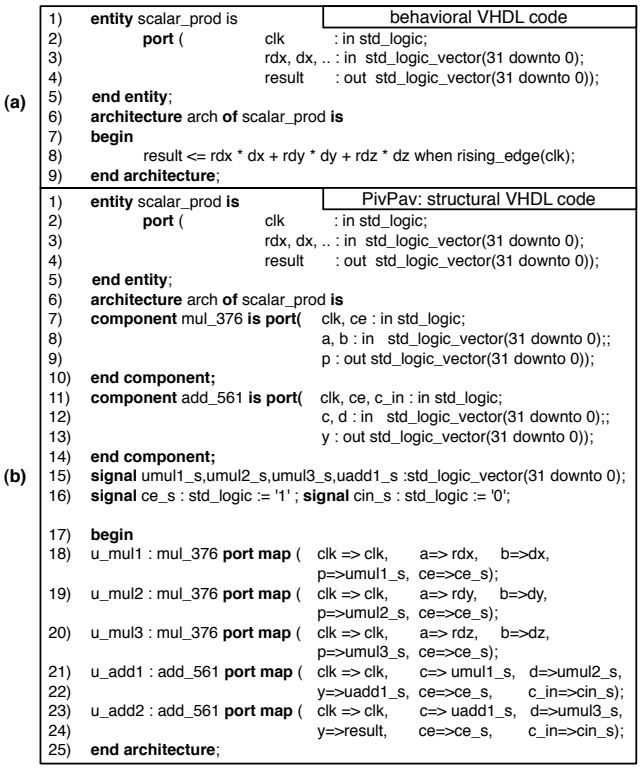


Figure 2: Data path generator

not need to re-generate or re-synthesize the basic operators, but the netlists for these operators are retrieved from the circuit repository.

Instead of manually running the FPGA tool flow for synthesizing the generated data path, the generator adds the data path to PivPav as a new customized circuit and makes use of the benchmarking facilities. To explore the impact of the FPGA tool flow configurations, we have implemented each circuit using six different optimization settings (design goals). Comparing the single-cycle implementation obtained from behavioral synthesis with the Pipelined(2) implementation reveals a speedup by a factor of 4.07 but comes at the price of increasing the latency to 13 cycles. The table row Pipelined(1) presents data for the same circuit built up from combinational components with pipeline registers inserted after each operator (latency = 1). While the latency increases from 1 to 3 cycles, the maximum operating frequency of Pipelined(1) is lower than for the Behavioral variant. The substantial differences in hardware resources are attributed to the fact that Pipelined(1) implements the functionality with LUTs rather than DSP blocks.

If we consider the availability of numerous metrics (in particular, area, speed, latency, and power consumption), we can see the full potential of PivPav as a backend for generating data paths for different scenarios, such as maximum performance, minimum power consumption, or minimum use of resources.

*Conclusions:* The PivPav API enables any tool to construct structural HDL code from a library containing thousands of operators. The automated construction of hardware is possible due to the fact that PivPav delivers detailed informations about each entity of a given operator. Further, the results from all steps in the FPGA implementation tool flow, for example the optimized netlist, are also available in the circuit library. Hence, no re-synthesis of the operators is required, which allows for rapid data path synthesis.

### C. Design Space Exploration

As PivPav combines a large library of circuits which are characterized by functional and non-functional metrics, the tool is well suited for design space exploration, where a rapid identification of the best candidate circuit respecting design constraints and goals is required. Typical scenarios are finding the fastest circuit of a given function that does not exceed a resource usage limit, or selecting the most energy efficient operators, i. e., balancing power consumption and performance. The applications presented in the previous case studies are actually exercising this design space exploration functionality of PivPav. Driven by optimization criteria, such as performance or resource constraints, the optimal circuits of the appropriate type (32 bit signed integer inputs) are extracted from the circuit library. An excerpt from a query to the circuit library for these criteria is shown in Table I.

One optimization parameter that FPGA developers are frequently trying to exploit is to reduce the precision of the data types in the design whenever possible to save resources and power and to increase the clock rate. The effect of this optimization, however, is not known until the optimized design is synthesized again. With PivPav it is possible to make a rapid estimate how the accuracy optimization will affect a particular operator, which allows for rapid exploration of the optimization design space.

An interesting possibility enabled with PivPav is to explore not only the design space for circuits, but to explore the design space of circuit generators. For example, it is possible to keep the function of the circuit fixed, but to use different circuit generators, such as Xilinx Coregen or FloPoCo, and to explore metrics of the generated circuits. Further, it is possible to keep the circuits fixed, but to vary the parameters of the FPGA CAD tools, for example, using different optimization methods and strategies, or target FPGA architectures. The data from these experiments can be used for answering questions like: Which FPGA family, device, and speed-grade can satisfy a given performance bound in the most cost effective way? Which CAD tool settings provide the best trade-off between runtime and performance? How did the quality of FPGA CAD tools change between different versions of the tools?

In the following example, we demonstrate the use of PivPav for exploring FPGA CAD tools. Figure 1 shows that the design space exploration application adds new circuits to the library, changes the settings of the CAD tool flow, and uses the benchmarking facilities to extract the resulting implementation metrics. For this example, we consider the circuits Behavioral, Pipelined(1), and Pipelined(2) that have been used in case study III-B. After adding these circuits to the library, we benchmark them with six different design goals. The results of this process are presented in Table II.

The table shows that PivPav collects the runtime and memory usage for each execution of the FPGA tool flow. Further, it collects the circuit characterization data for each FPGA CAD algorithm setting. The setting $b$ (minimum runtime) has almost the same runtime as the $a$ (balanced) settings, also the circuit characterization data for both of these settings are exactly the same. Settings $d$, $e$, and $f$ have the highest runtimes. This kind of explorations can lead to a better understanding of FPGA CAD tool flows and point out their strengths and weaknesses. The knowledge gained from these observations can be applied to improve the quality of PivPav's circuit library and consequently improve the results of the tools that use PivPav as a backend.

*Conclusions:* PivPav can be used to explore and find a set of circuit candidates matching given criteria for a specific FPGA device. In addition, it can be used to compare the behavior of FPGA CAD tools for different parameter settings or to compare different CAD tools and algorithms with each other. As PivPav stores all circuits in a common library, the results of all design space exploration tools will profit from new circuits being added to the library.

## IV. Implementation

A schematic overview of the architecture of the PivPav tool is presented in Figure 1. The tool comprises three main parts: [a] the circuit factory, [b] the benchmarking facilities, and [c] the circuit library. The *circuit factory* component is responsible for generating new circuits to be added to the PivPav framework. To this end, PivPav provides an interface to existing circuit generators, but also allows for adding custom circuits. The *benchmarking framework* automates the process of generating a circuit implementation using the appropriate FPGA CAD tool flow and extracting the metrics that characterize each circuit. The results of the benchmarking process, which includes currently more than 90 different properties and metrics for each circuit, are finally stored in the database of the *circuit library*. The primary use case for PivPav is to serve as a backend for higher level tools. These custom applications can access the data in the circuit library via a defined API.

PivPav is implemented primarily in the TCL and C++ programming languages. The choice of TCL is motivated by the fact, that it is currently the most widely used language for FPGA CAD tool scripting. In particular, the Xilinx design tools are scriptable with TCL, hence we use TCL for implementing the circuit factory and benchmarking components. The PivPav API and database access routines are implemented in C++, which makes it easy to use PivPav as a library from high-level tools through the C++ API. This is particularly important for our main use case of PivPav for data path generation in high-level compilation, because most compiler frameworks are implemented in C or C++.

| | Goal | computation time [secs] | | | | | memory usage [MB] | | | | circuit metrics | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FPGA Alg. | | xst | ngd | map | par | | xst | ngd | map | par | | | | | | | |
| | | | | | | Total | | | | | Lat. | Freq. | Power | FF | LUTs | Slices | DSP |
| | | | | | | | | | | | [cyc] | [MHz] | [mW] | | | | |
| Behavioral | a | 8 | 4 | 10 | 38 | 60 | 412 | 321 | 696 | 724 | 1 | 67.94 | 875.35 | 47 | 79 | 62 | 9 |
| | b | 8 | 3 | 8 | 35 | 54 | 411 | 321 | 698 | 725 | | 67.94 | 875.35 | 47 | 79 | 62 | 9 |
| | c | 7 | 3 | 37 | 55 | 102 | 411 | 322 | 915 | 733 | | 66.92 | 864.87 | 32 | 94 | 62 | 9 |
| | d | 8 | 3 | 37 | 54 | 102 | 411 | 322 | 913 | 735 | | 77.69 | 876.00 | 32 | 62 | 43 | 9 |
| | e | 7 | 3 | 46 | 93 | 149 | 411 | 322 | 936 | 699 | | 154.66 | 862.55 | 47 | 79 | 66 | 9 |
| | f | 8 | 3 | 45 | 56 | 112 | 411 | 322 | 941 | 726 | | 167.06 | 876.00 | 32 | 62 | 39 | 9 |
| Pipelined(1) | a | 14 | 5 | 13 | 53 | 85 | 443 | 340 | 722 | 804 | 3 | 53.41 | 889.61 | 372 | 2311 | 1314 | 3 |
| | b | 15 | 6 | 12 | 54 | 87 | 443 | 345 | 722 | 804 | | 53.41 | 889.61 | 372 | 2311 | 1314 | 3 |
| | c | 15 | 6 | 74 | 110 | 205 | 443 | 345 | 956 | 782 | | 60.90 | 897.54 | 372 | 2311 | 1329 | 3 |
| | d | 15 | 6 | 70 | 192 | 283 | 443 | 345 | 955 | 780 | | 53.83 | 2039.16 | 148 | 2311 | 1233 | 3 |
| | e | 15 | 6 | 59 | 147 | 227 | 443 | 345 | 952 | 734 | | 63.73 | 896.23 | 372 | 2311 | 1389 | 3 |
| | f | 15 | 6 | 88 | 249 | 358 | 443 | 345 | 949 | 791 | | 55.17 | 2044.65 | 184 | 2311 | 1249 | 3 |
| Pipelined(2) | a | 9 | 4 | 8 | 37 | 58 | 413 | 326 | 701 | 769 | 13 | 227.38 | 880.37 | 474 | 237 | 266 | 9 |
| | b | 9 | 4 | 8 | 36 | 57 | 413 | 327 | 701 | 770 | | 227.38 | 880.37 | 474 | 237 | 266 | 9 |
| | c | 9 | 4 | 46 | 66 | 125 | 413 | 327 | 919 | 737 | | **276.40** | 889.32 | 474 | 237 | 356 | 9 |
| | d | 9 | 4 | 39 | 75 | 127 | 413 | 328 | 918 | 738 | | 194.33 | 2031.40 | 250 | 237 | 169 | 9 |
| | e | 10 | 4 | 55 | 64 | 133 | 413 | 328 | 943 | 732 | | 263.37 | 890.63 | 472 | 239 | 372 | 9 |
| | f | 9 | 4 | 50 | 79 | 142 | 413 | 328 | 943 | 731 | | 179.05 | 2034.84 | 248 | 239 | 187 | 9 |

Table II: Measured characterization data for three implementation variants of the example shown in Figure 2. Design Goal legend: a = balanced, b = minimum runtime, c = power optimization, d = timing performance optimization with IOB, e = timing performance optimization with physical synthesis, f = timing performance optimization without IOB.

While PivPav has been mainly developed from scratch, we leverage two existing frameworks in our work. For the database, we use the Sqlite3 database [8], which is a full-featured relational database with SQL support, but is very lightweight and does not need any server installation. Further, we leverage the FloPoCo circuit generator for two purposes. First, we use FloPoCo as an external tool as one source for generating circuits in the circuit factory component. Second, in the PivPav API we reuse parts of FloPoCo as a library for conveniently generating VHDL source code from C++.

To encourage other researchers to use and contribute to PivPav, we release the tool as open-source software under the GNU GPL license. The source code is available at: `http://github.com/pc2/PivPav`. Additionally, we provide basic documentation on using PivPav and its API on the PivPav website: `http://pc2.github.com/PivPav/`.

Discussing the implementation of PivPav in full detail is out of the scope of this paper. Instead, we will in the following highlight some selected design and implementation aspects of the circuit factory, the benchmarking framework, and circuit library. Finally, we will illustrate the use of PivPav's API with an example. To learn more about the implementation, we refer the reader to the source code and the included documentation.

### A. Circuit Factory

The purpose of the circuit factory is to provide circuits for populating the circuit library. PivPav does not provide its own circuit generator, but relies on existing circuit generators. Currently, PivPav supports the Xilinx Core Generator (Coregen) and the open-source FloPoCo tool as configurable generators for arithmetic circuits. Additionally, it is possible to import arbitrary circuits as netlists, provided that a VHDL declaration for the interface is available, or as pure VHDL if the source codes of the circuits exist.

The circuit factory provides tools for creating new circuits with the commands `gen_flopoco.sh` for the FloPoCo generator, or `gen_coregen.tcl` for Coregen. For Coregen circuits, this tool also implements a unified command-line interface that allows for specifying the circuit parameters, such as bit-width, data types, or pipeline stages, in a consistent way for all relevant arithmetic cores. The resulting parameters are exported to a `XCO` configuration file that is used by Coregen as specification for the circuit generator. This unified interface shields the user from the inconsistencies in the naming of core configuration parameters in Coregen, which, for example, uses either the parameter name `latency` or `pipestage`, depending on the circuit, for configuring the same property. For initially populating the library, PivPav provides scripts in the `variants` directory that perform a parameter sweep and generate a whole range of circuits for a configurable set of operations, bit-widths, and latencies.

Once a circuit is generated, its top VHDL entity is automatically parsed and a set of generator-specific rules is applied to identify the semantics and data types of the circuit input and output ports. This process assigns each port a specific role, for example, an input port can be an operand port, a reset signal, or a clock signal. This meta information about the circuit is also stored in the circuit library and is used for example when constructing data paths from operators as described in case study III-B. For custom circuits, which can be added with `addCustomCircuit.tcl` to the library, the tools try to guess the semantics of the input and output ports by applying a set of configurable pattern matching rules. For instance, the clock port is searched under the names of `clk` and `clock`, and the clock-enable port is searched under the names `ce`, or `clk_ce`. If the whole entity cannot be successfully parsed in this manner, the script enters an interactive mode and lets the

designer choose the semantics of the ports. The parsing of circuit interfaces is implemented in the `parse_api.tcl` and `ports_api.tcl` tools.

### B. Benchmarking Framework

The purpose of the benchmarking framework is to run the FPGA CAD tool flow on the circuits generated by the circuit factory in order to obtain various kinds of implementation-related meta information, such as the maximum operating speed, or the estimated power consumption. Currently, our benchmarking framework supports the Xilinx ISE Foundation FPGA CAD tool flow.

The benchmarking process is implemented in the `bench.tcl` tool. The first step is to extract the netlist and VHDL files that define the circuit from the library. If required, combinational circuits are instantiated using a wrapper, which is a precondition for specifying timing constraints for the implementation (see IV-D for more details). After the tool has used Xilinx ISE's TCL interface for setting up a new project and adding the required design files, the tool flow parameters are configured according to the *design goals*. The concept of design goals is an abstraction we have introduced in our tool for encapsulating one particular set of optimization parameters for the FPGA tool flow. We provide a set of six predefined design goals, for example, optimizing for maximum performance with and without physical synthesis, or minimizing the runtime of the design tools. The design goals are implemented as a set of TCL commands that are supplied to ISE, for example, power reduction is enabled by the command `project set "Power Reduction" "true" -process "Map"`. More informations about these statements can be found in [9] and [10, chapter 3]. In a final step, the target FPGA family, device, speed grade and package is configured. After the setup of the design is completed, `bench.tcl` launches the FPGA CAD tool flow and monitors the execution of the tools searching for errors and warnings. Additionally, the execution time for each step in the tool flow is determined.

If the implementation completes successfully, a set of metrics is extracted from the results with the `reports.tcl` tool that is available in the `reports` directory. The design metrics are extracted by applying pattern matching to the implementation log files. For each metric, a regular expression is declared that specifies what pattern is searched and under what name the result is stored in the circuit library. For example, the pattern for extracting the number of lookup tables from the report of the *map* process of the tool flow and storing it under the name of `n_lut` is specified in `report_map.tcl` as: {`Number of 4 input LUTs:\s+(\d+)`} `"n_lut"`. This mechanism allows for configuring which metrics are stored in the library in a convenient, declarative way. Finally, when all circuit metrics have been gathered, the metrics are stored along with the circuit in the library, where they can be later extracted by other tools or re-evaluated with the different FPGA CAD settings or tool flows.

### C. Circuit library

The circuit library is essentially a database that stores all circuit source codes and netlists, their parsed VHDL interface, and the meta data extracted in the benchmarking framework. We use the Sqlite3 database for this purpose, since it has native interfaces to the TCL and the C/C++ programming language.

### D. Building Tools using the PivPav API

While PivPav can be used as a standalone tool for circuit benchmarking, the strength of the tool is the possibility to interact with other tools by means of the API. This API can be considered the most important feature of PivPav and has been used, for example, to build the tools shown in the case studies. Our primary use case that guided the development of PivPav is to use the tool as a circuit library for high-level synthesis. High-level synthesis requires to integrate two domains, the software domain, which is reigned by a compiler framework, and the hardware domain that controls the FPGA CAD tool flow. The software compilers are typically implemented in C++, while the CAD tool flows are natively scriptable in TCL. The PivPav API allows for interacting with the circuit library for both domains and from both programming languages.

The basic way for obtaining information from the circuit library is to directly use SQL queries, which ensures that custom queries for any purpose can be built. For example, this allows a tool to query the circuit library for the best candidate circuit under given constraints, where the constraints can formulated using a choice of more than 90 different metrics which are currently available in the library. Additionally, PivPav comes with a set of predefined queries that are frequently used, for example for querying the library for the fastest or smallest circuit of a given function, avoiding the need for writing custom SQL queries. For both cases the circuit library will receive an SQL query and will return results similar to Table I.

The C++ API is particularly important for the high-level tools. PivPav includes libraries which allow to create and to manipulate VHDL code in the C++ language. These libraries are based on code from the FloPoCo project and allow for generating VHDL code very effectively. While presenting the full API of the VHDL code writer is out of the scope of this paper we show an example for the *wrapper* tool, which was written in only 70 lines of code and which emphasizes the use of the API. The wrapper tool is used in PivPav's benchmarking facilities for wrapping a combinational circuit with input registers, which is a precondition for applying timing constraints for the FPGA tool flow.

Figure 3 presents an overview of the wrapper, that is composed of four libraries: `libsql` which is responsible for querying the circuit library, `libhw_write`, `libgetOperator`, and `libwrapper.cpp`. `libhw_write` uses modified code from FloPoCo [2] and makes use of FloPoCo's Operator and Signal classes for generating VHDL from C++. The `libgetOperator` library uses `libsql` in order to obtain informations about the circuit from the library. This information is used to instantiate an Operator object in the `lib_hwwrite`

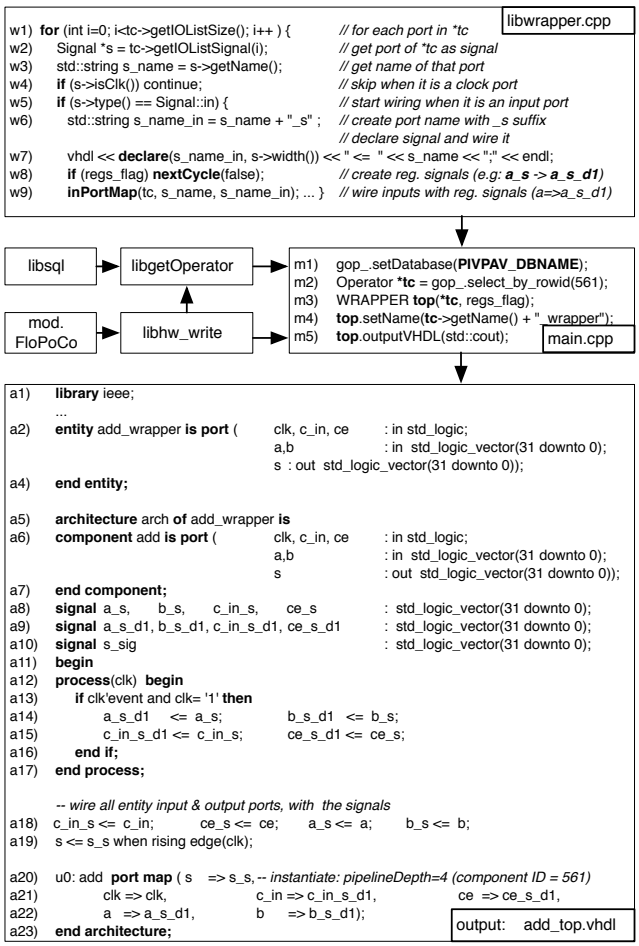


Figure 3: Impl. of the *wrapper* tool with the PivPav API.

namespace, which is shown in Figure 3 at line `m2` where `getOperator` instantiates the circuit (ID 561) from the library. This is an essential feature of the API that is simple to use but very powerful, because it allows to connect the hardware and the software domain. The digital circuit with its metadata informations is transformed into a FloPoCo Operator instance, which is a pure software object. This transformation allows the high-level tools to use, transform, modify, or connect circuits, without ever leaving the software domain boundaries. After transforming the circuit to an Operator the `libwrapper.cpp` library is used to register the input and the output ports (line `m3`). The part of the source code responsible for inserting registers at the inputs of the instantiated circuit is shown in the top part of Figure 3.

Line `w1` iterates over the IO ports of the Operator and line `w2` assigns the given IO port to a Signal object, which is equivalent to the entity's port in the hardware domain. The name of the signal is obtained in line `w3`. Since the clock signal does not have to be registered we skip it in line `w4`. We select only the input signals in line `w5` and in line `w6` we duplicate the signal name and add a `_s` suffix to it. Line `w7` uses the API's `declare()` function which generates lines `a8` and `a18`. Next, the `nextCycle()` function in line `w8` creates registered signals with the help of a VHDL process, which translates to the VHDL code in lines `a9` and `a12-17`. The `inPortMap()` function in line `w9` instantiates the circuit component in lines `a21-22`. Finally, the top entity name (line `a2`) is changed in line `m4` and the final VHDL code is printed with line `m5`.

As presented in this detailed example, the API allows for writing very dense code that mediates between the digital hardware domain and software object instances in a clear way. The same C++ API was also used to implement the data path generator described in the case study III-B.

## V. Conclusions

In this paper we have introduced the PivPav tool, which combines a circuit factory, benchmarking facilities and a circuit library that can be accessed with an API. PivPav's functionality can be used in other tools by using this API or by directly accessing the circuit library that is stored in an SQL database. We have presented three case studies that illustrate the use of PivPav with important, real-world problems. These application demonstrate the usefulness of the tool in three major areas of the custom computing domain: instruction-set extensions, data path generation, and design space exploration. The tools developed in these case studies show that PivPav is a self growing system, where the results obtained from one of the tools can be reused by other tools relying on the same circuit library. This increases the power of PivPav with every circuit that is added to the library. In our research, PivPav has proven itself as a useful tool and we plan to leverage it for our future work. By releasing PivPav as open-source software, we encourage other researchers to use the tool for their purposes and to contribute to the development of PivPav.

## References

[1] *CORE Generator Guide*, Xilinx, San Jose, CA, USA, 2010.
[2] C. K. Florent de Dinechin and B. Pasca, "Generating high-performance custom floating-point pipelines." IEEE Computer Society, August 2009.
[3] *Megafunction Overview User Guide*, Altera, San Jose, CA, USA, 2010.
[4] *GRLIB User's Manual*, Aeroflex Gaisler, Goteborg, Sweden, 2010.
[5] M. Grad and C. Plessl, "Woolcano: An architecture and tool flow for dynamic instruction set extension on Xilinx Virtex-4 FX," in *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. CSREA Press, Jul. 2009, pp. 319–322.
[6] M. Tofall, "Compiler for a custom instruction set CPU," Master's thesis, University of Paderborn, Germany, Computer Engineering Group, Paderborn, Germany, Oct. 2009.
[7] *Floating-Point Operator v5.0*, Xilinx, San Jose, CA, USA, June 2009.
[8] M. Owens, *The Definitive Guide to SQLite*. Apress, May 2006.
[9] "ISE TCL quick reference card," Xilinx, San Jose, CA, USA, 2007.
[10] *Development System Reference Guide*, Xilinx, San Jose, CA, USA, 2010.