PyGears: A Functional Approach to Hardware Design

Bogdan Vukobratović
Novi Sad, Serbia
bogdan.vukobratovic@gmail.com

Andrea Erdeljan
Faculty of Technical Sciences
University of Novi Sad
Novi Sad, Serbia
andrea.erdeljan@uns.ac.rs

Damjan Rakanović
Faculty of Technical Sciences
University of Novi Sad
Novi Sad, Serbia
rdamjan@uns.ac.rs

***Abstract*—In this paper we propose a new hardware design methodology called Gears, and we introduce PyGears, a Python framework that facilitates designing hardware using the Gears methodology. Gears builds on top of the RTL methodology and focuses on hardware module composability, hence improving design reuse, scalability and testability. Gears methodology proposes building complex digital systems from small functional units that communicate exclusively via simple handshake interface called Data Transfer Interface (DTI). Such units form a category (from the mathematical field of Category Theory), and Gears then provides practical means for composing such units to implement a complex functionality using concepts from the Category Theory (like algebraic types and functors). On the other hand, PyGears helps describe these abstract composition operations in a way that is easy to read and debug, it compiles the design described in Python to SystemVerilog and allows simulating the design.**

***Index Terms*—RTL, HDL, Python, HLS, functional**

I. Introduction

RTL methodology is still a primary method for describing digital hardware systems, usually via some variation of the Finite State Machine with Datapath (FSMD) model [1]. The FSMD intermediate model provides a way of translating sequential algorithms into a hardware description consisting of: a dataflow which implements data transformations, and a control flow (described using a FSM) which controls under which circumstances the transformations are applied. However, for real world examples, the obtained control flow FSM can be quite complex, and RTL methodology offers very few guidelines on how to efficiently refactor the design into smaller manageable units. Manipulating a complex FSM (like debugging, adding a new feature, pipelining, etc.) can thus offer a significant challenge. Furthermore, complex modules are hard to reuse and thoroughly test.

There are many attempts at providing an alternative to the RTL [2]. They usually focus on a specific design domain like: streaming systems [3] [4], or describing the design in form of concurrent FSMs [5] via so-called guarded atomic actions. These tools can be valuable when the task at hand matches the domain they support, but they offer no benefits outside of it. For example, dataflow dominant designs, like DSP algorithms, are not easily described using guarded atomic actions, and stream programming model is not well suited for a control flow heavy design like a microprocessor.

Next, there are numerous methods that propose describing hardware in a procedural languages like C, and then offer automated tools to synthesize the procedural description into a standard HDL [6] [7] [4]. These have only had a limited success and are also usually targeted for a specific domain, like streaming applications.

In this paper, we introduce Gears, a novel hardware design methodology that aims to complement the traditional approach by imposing additional constraints on the hardware modules being designed. Authors have recognized that the lack of composability of the modules generated by the traditional methodologies significantly increases the complexity of the hardware design in general. Gears requires the use of a simple handshake interface called DTI for all communication between modules, which allows for them to be viewed as mathematical objects from Category theory [8], which in turn provides a rich set of tools for composing such modules. Ease of module compositions allows for factoring the design into smaller units, which are easier to write, test and reuse, which in turn enables forming comprehensible, well-tested libraries. This finally leads to significantly reduced efforts when designing hardware with such libraries.

Regarding the hardware-description languages, Verilog and VHDL are still by far the most commonly used. These languages were originally designed to develop hardware simulation models, and only later did hardware synthesis tools provide support to generate hardware from subsets of these languages. Additionally, these languages have very few means of combination and abstraction compared to modern programming languages, which further reduces the composability of described hardware modules. SystemVerilog was created later as an extension of Verilog, but mainly improved upon its verification capabilities.

In an attempt to provide an alternative to the traditional hardware-description languages, many different programming languages have been re-purposed for the task of the hardware design, like: Scala [9] [10], Haskell [11], Python [12], etc. While these languages offer higher level constructs that do facilitate certain aspects of describing the hardware, they still rely on the traditional RTL methodology, and as such do not raise the abstraction level at which the hardware is designed.

The Gears methodology can be exercised with any

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

hardware-description language, with some of them being more suitable then others, like SpinalHDL [10] with its generic Stream interfaces which are equivalent to the DTI. Nevertheless, we developed PyGears, a Python framework for hardware description that is specifically designed to express composition of the modules adhering to the Gears methodology in a most elegant and natural way.

II. Gears Methodology

A. DTI interface

Traditional methodologies put no constraints on the interfaces a hardware module can implement, which results in hardware modules that are hard to connect to each other, i.e compose. Even though there are many on-chip standardized interfaces used in industry today, like: AMBA [13] (AXI, AHB, APB, etc.), Avalon [14], etc., they are usually used to connect large hardware modules, i.e. IP cores, when developing System on Chip (SoCs). Gears methodology pushes the interface standardization all the way down to the smallest functional units (like registers, MUXs, counters, etc.), by forcing each unit to implement a simple synchronous handshake interface (somewhat similar to the AXI4-Stream interface) called DTI.

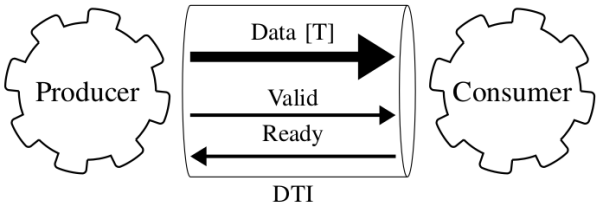

Fig. 1. DTI - Data Transfer Interface

DTI interface is used to connect two hardware modules called a producer and a consumer, and transfers data from the producer to the consumer. It consists of three signals as shown in the Fig. 1:

- Data - Variable width signal, driven by the producer, which carries the actual data.
- Valid - Single bit wide signal, driven by the producer, which signals when valid data is available on Data signal.
- Ready - Single bit wide signal, driven by the consumer, which signals when the data provided by the producer has been consumed.

The protocol employed over the DTI interface was designed to help a designer reason about the hardware system at a higher level of abstraction, namely in terms of the data exchange, not in terms of manipulating the individual signals. Handshake mechanism helps avoid race-conditions and ensures the proper transfer of data between asynchronous modules. The modules that adhere to the DTI protocol are called *gears*. Communication over DTI entails the following procedure shown on Fig. 2 in a form of a waveform:

1. The producer initiates the data transfer by posting data on the Data signal, and rising Valid signal to high, as seen in cycle 1, 6 and 7.

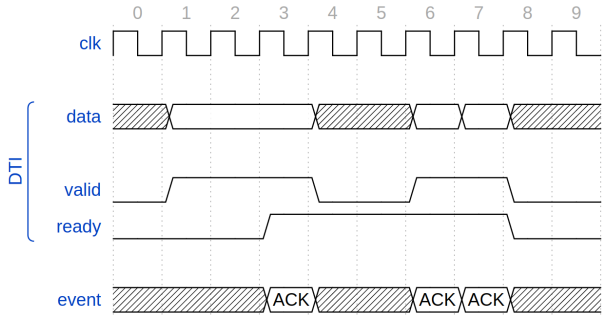

Fig. 2. Waveform describing the DTI protocol

2. The consumer can start using the input data in the same cycle the Valid line went high.
3. The consumer can use its input data driven by the producer for internal calculations for as many cycles as needed. For example in cycles 1-3.
4. When consumer realizes that it is the last cycle in which it needs the input data, it raises the Ready signal to high (cycles 3, 6 and 7, marked also as ACK). On the edge of the clock if both Valid and Ready signals are high, it is said that the consumer acknowledged/consumed the data, or that the handshake has happened. This signals the producer that in the following cycle new data transfer can be initiated, or Valid signal can be set to low (cycles 4 or 7), which pauses the data transfer.
5. After initiating the transfer, producer needs to keep the Data signal unchanged and the Valid signal high until the handshake occurs, as seen in cycles 1-2.
6. The producer can keep Valid signal low for as many cycles as needed, which blocks the consumer if it is waiting for new input data, as seen in cycles 6-7.
7. There must be no combinatorial path from Ready to Valid signal on the producer side. In other words, the producer should not decide whether to output the data based on the state of the consumer, but only based on its own inputs and internal state.
8. The consumer may decide whether to acknowledge the data based on the state of the Valid signal, i.e. there may exist a combinatorial path from Valid to Ready signal on the consumer side.

Any composition of gears again yields a gear which obeys all the listed rules, i.e. gears are closed under composition, where by the gear composition we basically mean connecting two gears via DTI interface. This means that composing gears is predictable in many ways, and having rich and verified low level library of gears, translates to reliable description of higher level modules, where many (especially synchronization) errors are avoided by design.

B. Data types

Since gears are closed under composition and their composition is associative (it does not matter how the gears are grouped, only the sequence in which they are connected via

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

DTI), they form a category in the Category theory. In order to enrich this category, additional piece of information is associated with each DTI port, namely its data type, which then also determines the width of its Data signal. This way, a category is formed, whose objects are data types and whose morphisms are gears as shown in the Fig. 3. The figure shows how two example gears: f with the input interface type T_1 and the output interface type T_2, and g with the input interface type T_2 and the output interface type T_3, can be composed to form a new gear $g \circ f$.

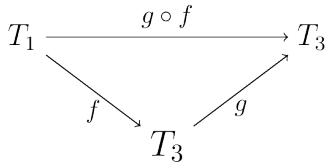

Fig. 3. Gear composition diagram in terms of the Category theory

It is important to note that transmission of a single instance of a data type over DTI can span multiple clock cycles. For example a data type can be defined that represents transactions of length 8, where each item is a 16-bit integer, where only one 16-bit item is transmitted per clock cycle. Which data types are supported and how they are encoded on a DTI data signal is not handled by the Gears methodology, but is implemented in PyGears as it is described in Section III-C.

Mapping of the gear composition onto a category, provides a designer with a rich set of tools from the Category theory, like the use of the algebraic data types and functors, which are heavily used when describing hardware with PyGears. Algebraic data types show how basic data types can be combined in a meaningful way, and functors offer a way of using gears which operate on basic data types in contexts where complex data types are present. This way, the Gears methodology maximizes module reuse, which in turn minimizes the design and debugging efforts. Upon introducing data types for the interfaces, it is useful to regard gears as functions, gear connection as function composition and exchanging data as function calls, which significantly raises the level of abstraction at which the system is designed.

C. Gears purity

As discussed in the Section I, the FSMD model is often used for translating sequential algorithms into hardware, which produces complex control flow FSMs for any sufficiently complex real world example. Number of possible walks through the FSM the designer needs to reason about, akin to the cyclomatic complexity in computer science, rises rapidly with the number of allowed transitions and the length of the walk. It is shown in [15], that the upper bound for the number of walks can be calculated as:

$$N_k <= \sum_{x \in V} d_x^k \tag{1}$$

where, k is the length of the walk, N_k is the number of possible walks of length k, V is set of all FSM states and and d_x is a number of transitions from the state x. Even worse, when composing two modules with FSMs, the number of transitions is effectively the product of the number of transitions for each individual module, hence the total number of walks the designer needs to be aware of is the product of the number of possible walks for each of the two modules.

Gears methodology tries to alleviate this by introducing a concept of "pure gears" and advocating their heavy use. Pure gear is a module that has a well defined initial state, and always returns to this state upon calculating its output and consuming its input data. Such gears are more predictable and easier to reason about when composed together. Easiest examples of pure gears are the ones which do not have a state of their own, i.e. the ones described using combinatorial logic only. As mentioned in the Section II-B however, some data instances transmitted over DTI can span multiple clock cycles, hence the gear that works on such data will require multiple cycles for its computation. Such a gear can still be considered pure as long as it returns to its initial state upon receiving whole data instance, i.e. at the end of the input transaction.

III. PyGears framework

A. Gear definition

Although a design process in any hardware-description language could benefit from following the Gears methodology, we designed a Python framework called PyGears to express gear composition in a most elegant and natural way, i.e. in terms of function compositions. Choosing Python as a description language has many benefits:

- Python has a clean, understandable syntax which makes a hardware description written in it easy to read and debug,
- there are many libraries available for Python which can be used for simulation,
- Python makes for a powerful, high-level compile-time preprocessor for a hardware description,
- Python has many options for connecting and embedding third-party modules, which makes PyGears easily extensible

A gear is defined in PyGears using a function construct in the following way:

```
@gear
def gear_name(
        in1: T1, ..., inN: TN,
        *,
        p1=dflt1, ..., pM=dfltM
        ) -> ReturnType:

    Function body: gear implementation
```

Python decorator statement `@gear` informs PyGears that what follows is not a regular Python function definition, but a gear definition. `def` is a Python keyword for function definition, and the `gear_name` is where the name of the gear is stated (akin to module/entity names in Verilog/VHDL), and it will be used later to make instances of the gear.

In brackets, the input DTI interfaces and compile-time parameters are declared, where the character "$\star$" delimits

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

between the two. Each input interface: `in1`, ... `inN`, can have a type declared too: `T1`, ... `TN`. Input types will be used by PyGears at compile time to perform type checking, to setup the gears which are polymorphic, and to automatically infer some connectivity logic between the gears to facilitate the composition. A gear can optionally support compile time parameters, `p1`, ... `pM`, with their default values `dflt1`, ... `dfltM`, that can be used to configure the gear instance. Finally the `ReturnType` specifies the types of the output interfaces.

The body of the function is used to describe the gear implementation in one of the two ways depending on the type of the gear being described. Gears that implement the smallest functional units belong to the first type, and are described in SystemVerilog following the Gears methodology. These gears do not have to have a description in Python, i.e. the body of their function definitions can be left empty since they are fully defined by their SystemVerilog descriptions.

Gears that are described in terms of the composition of lower-level gears are of the second type, and are analogous to the hierarchical modules of traditional HDLs. They do not have a SystemVerilog implementation because it is generated automatically by PyGears given their Python description.

B. Gear instantiation

Once defined, a gear can be instantiated as many times as needed in the design using different set of input interfaces and parameter value combination. Gear instantiation is written in form of a Python function call by supplying the input interfaces (`in1`, ..., `inN`) and parameter values (`p1=val1`, ..., `pM=valM`) as function arguments.

```
outputs = gear_name(in1, ..., inN,
                    p1=val1, ..., pM=valM)
```

For a gear with a single output interface, the function call returns a Python object that represents that output interface. For a gear with multiple outputs, a tuple of interface objects is returned. Returned output interface objects can then be supplied when instantiating other gears, which then establishes connections between the gear instances.

A graph of interconnected gear instances created using Python function calls in a manner described above can then be translated to synthesizable SystemVerilog code by PyGears. Furthermore, PyGears features a simulator built on top of the Python *asyncio* framework [16] that can connect to an external HDL simulator (like Verilator [17]) to simulate the design together with its verification environment. Components of the verification environment can be also written as gears with their functionality described in pure Python, but the details of this process are out of the scope of this paper.

C. Data types

PyGears features several basic data types, like the unsigned and signed integers of generic width, namely `Uint[W]` and `Int[W]`. However, the real power the types offer comes from combining these basic types to form complex data types, i.e. algebraic data types. PyGears supports the following complex data types:

- `Tuple[T1, T2, ..., TN]` - is a heterogeneous container that corresponds to the *product* type in the Category theory and is akin to structs and records of standard HDLs. The types of it fields are defined by the types: `T1`, `T2`, ..., `TN`.
- `Union[T1, T2, ..., TN]` - corresponds to the *co-product* type in the Category theory, but it is also known as a *sum* type. `Union` can represent only one of its types (`T1`, `T2`, ..., `TN`) at a time. It is somewhat similar to unions in other languages with an exception that PyGears `Union` carries the information about which of the types it currently represents together with the data.
- `Queue[T]` - is a data type which describes a transaction and spans multiple cycles. Together with the data, it carries additional information which flags the last data item within a transaction.

PyGears framework also features a library of gears that can be used off the shelf, majority of which are polymorphic in the sense that they can adapt their inner operation to the types of the interfaces connected to their inputs. One example is the builtin `fmap` gear, which allows connecting interfaces with complex data types to gears that operate on some part of that type. This all means that selecting interface data types is an important step in the design process, since much of the hardware description will be automatically generated based on the type selection.

IV. An example: Moving Average Filter

A. Implementation

To illustrate the Gears methodology and the PyGears framework we show how a moving average filter can be implemented. The moving average filter is a simple Low-Pass FIR filter commonly used for reducing random noise while retaining a sharp step response. The filter operates by averaging a number of points from the input signal to produce a single point of the output signal. In other words it performs a convolution of the input sequence $\mathbf{x}$ with a rectangular pulse of length M and height $1/M$ as:

$$y[i] = \frac{1}{M} \sum_{j=0}^{M-1} x[i+j] \tag{2}$$

The simplified block diagram of the developed gear is given on Fig. 4.

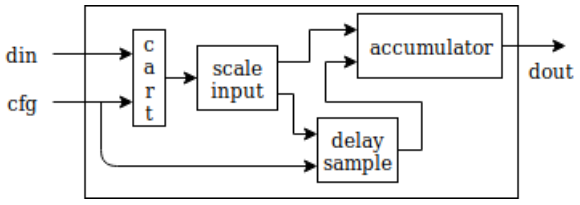

Fig. 4. Block diagram of the moving average gear

The filter has two input interfaces (one used for configuration and the other for data) and has a single output

interface. As with every gear, the interfaces are typed. The configuration carries two values: averaging coefficient and the size of the window, and is represented as a `Tuple` data type. The second input is used for streaming the data and is represented as a `Queue` data type. Compile time parameters of the moving average gear include the data width, shift amount and the maximum filter order. The interface definition of the `moving_average` gear using PyGears is given below. Since this is a hierarchical gear the output interface type is determined by the return statement and need not be specified.

```
@gear
def moving_average(
    cfg: Tuple[{'avg_coef'  : Int['W'],
                'avr_window': Uint['W']}],
    din: Queue[Int['W']],
    *,
    W=b'W', shamt=15, max_filter_ord=1024):

    scaled_sample = cart(cfg['avg_coef'], din) \
        | fmap(f=scale_input(shamt=shamt, W=W),
               fcat=czip)

    delayed_din = delay_sample(
        scaled_sample,
        cfg['avr_window']
        W=W,
        max_filter_ord=max_filter_ord)

    return accumulator(
               scaled_sample, delayed_din, W=W)
```

The filter operates as follows. Each data sample received at `din` input interface is first scaled by the averaging coefficient received at the `cfg` input interface. Since each element of the `Queue` needs to be multiplied, we first create a `Queue` of `Tuples` (more precisely `Queue[Tuple[Int['W'], Int['W']]]`), whose each item is a following pair of values: (averaging coefficient, input sample) represented by the `Tuple` type. This is done by replicating the averaging coefficient (`cfg['avg_coef']` in the code) for each data sample with the builtin `cart` gear, which automatically performs replication based on its input data types. Data formed in this way are then sent to the `scale_input` gear which multiplies the two elements of the pair and shifts the resulting data to restore the fixed-point format. In PyGears the function composition and thus the connection between the gears can be described using pipe "|" operator. The `scale_input` gear operates on the `Tuple` data types, not on the `Queues` of `Tuples` output by the `cart` operation. In order to compose these gears nevertheless, a functor mapping can be utilized implemented by the `fmap` gear. The `fmap` gear will send each element of the input `Queue` to the `scale_input` gear, and then pack its outputs again in a `Queue` data type. Usage of this functor allows `scale_input` to be an independent gear with a single responsibility, which can be easily reused in multitude of contexts. Functors are powerful patterns for gear composition that significantly improve possibilities for gear reuse. There is one functor for each complex data type. Functors allow for gears that operate on simpler data types to be used in context where a more complex data type is needed.

Each new sample scaled in this way is then added to the window sum. In order to generate a new window sum from the previous window sum, the first sample of the previous window needs to be subtracted from the accumulated sum since it is not in the window any more. The accumulation takes place in the `accumulator` gear, while the `delay_sample` gear is used to provide the samples to be subtracted from the sum at appropriate times, as given in the definition of `moving_average`. Since the `scaled_sample` interface needs to be connected to both the `accumulator` and the `delay_sample` gears, additional data broadcasting logic is needed to ensure the correct synchronization between the gears. In PyGears this is done automatically.

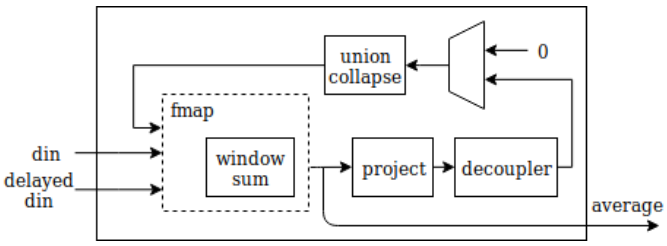

Fig. 5. Accumulator gear block diagram

The `accumulator` gear, whose block diagram is shown in Fig. 5 and its PyGears implementations shown below, contains a feedback loop that cannot be described as a plain gear composition since it forms a cycle. This cycle needs to be cut at one spot, described as the gear composition, and then joined together. The `prev_window_sum` interface is first defined without its producer gear and passed to the `window_sum` gear, then later connected to the output of the composition of the `priority_mux` and `union_collapse` gears.

The `window_sum` gear calculates the output average for one input sample at a time, but the data received at `din` is a `Queue` of samples, so an `fmap` needs to be used to connect the two. This is similar to how an `fmap` was used to connect the `scale_input` gear in `moving_average`. The result of the `window_sum` calculation is both sent to the output and used to form the `prev_window_sum`. The `average` interface is first connected to the `project` and `decoupler` gears, which discard the `Queue` (transaction) information and register the data. The `priority_mux` gear and the constant source of zeros: `Int[W](0)`, are used to either pass a zero value (for the first sample) or the value from the `average_reg` interface.

```
@gear
def accumulator(din, delayed_din, *, W):
    prev_window_sum = Intf(Int[W])

    average = din \
        | fmap(f=window_sum(prev_window_sum,
                            delayed_din),
               fcat=czip)

    average_reg = average \
        | project \
        | decoupler

    prev_window_sum |= priority_mux(average_reg,
```


This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

```
                                        Int[W](0)) \
        | union_collapse

    return average
```

The `window_sum` gear maintains the current window sum by adding a new input sample and subtracting the sample that is no longer in the averaging window, and its implementation in PyGears is shown below. All major arithmetic operators are supported by PyGears.

```
@gear
def window_sum(din, add_op, sub_op):
    return din + add_op - sub_op
```

B. Results

In this chapter we provide comparison between: PyGears, Vivado HLS and RTL implementations [18], in terms of the utilization and maximum attainable frequencies. Based on the python description of the `moving_average` gear, PyGears generates a SystemVerilog description. All implementations of the developed IP core were done using Xilinx's Vivado 2018.2 tool, with Zynq-7020 as the target FPGA device. The most interesting implementation results, regarding used hardware resources for the sample width of 16 bits (`W = 16`) and the maximum filter order of 1024, are presented in Table I.

TABLE I
FPGA RESOURCES REQUIRED TO IMPLEMENT THE MOVING AVERAGE CORE

Implementation	LUTs	FFs	BRAMs	DSPs	Fmax [MHz]
PyGears	102	91	0.5	1	168.60
RTL	63	58	0.5	1	155.95
Vivado HLS	248	183	0.5	1	181.79

As expected, the RTL implementation is the most efficient regarding the resource utilization since it builds the most cohesive but also the most coupled system. Nevertheless, PyGears implementation strikes an excelent balance between the RTL and HLS, by providing a convinience of writting at a high level of abstraction without a significant drop in performance. Moreover, the HLS implementation failed to achieve the desired throughput of 1 sample per clock cycle despite the optimization directives that were provided. That is, PyGears offers better controllability over the final result than HLS, which allows achieving performances closer to RTL.

V. Conclusion

In this paper, we presented a novel hardware design methodology called Gears to help with the composability issues present in traditional methodologies. By providing a system for composing modules at all levels of hierarchy, Gears offers a possibility to implement complex hardware systems from small units. Small modules with a single, well-understood functionality are easier to understand, test, debug, maintain and most importantly: reuse. With increased module reuse capabilities, one can focus on building a library of well-tested, well-understood modules, which then reduces time to market and minimizes debugging efforts.

Next, the DTI protocol forces local synchronization between the modules, which often helps avoid complex global control FSMs. Although having a handshaking interface at low level can seem as an overhead in terms of the latency and logic utilization, the DTI protocol was designed to minimize this overhead, and to allow modern hardware synthesis tools to remove any unnecessary handshaking logic.

Finally, we introduced a python framework called PyGears that provides clean, high-level syntax for the gear definition and composition. It also features a simulator and translates hardware description in Python to SystemVerilog. PyGears framework and hardware libraries are being made available as an open-source project under the MIT license which permits commercial usage, available at: https://www.pygears.org.

References

[1] Pong P Chu. *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. John Wiley & Sons, 2006.

[2] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, and others. *A survey and evaluation of FPGA high-level synthesis tools*. *IEEE TCAD*, 35(10):1591–1604, 2016.

[3] Jocelyn Sérot, François Berry, and Sameer Ahmed. *CAPH: a language for implementing stream-processing applications on FPGAs*. In *Embedded Systems Design with FPGAs*, pages 201–224. Springer, 2013.

[4] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alexandru Nicolau. *SPARK: A high-level synthesis framework for applying parallelizing compiler transformations*. In *VLSI Design, 2003. Proceedings. 16th International Conference on*. 2003.

[5] Rishiyur Nikhil. *Bluespec System Verilog: efficient, correct RTL from high level specifications*. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, 69–70. IEEE, 2004.

[6] *Vivado High-Level Synthesis*. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. Accessed: 2018-08-12.

[7] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. *Designing modular hardware accelerators in C with ROCCC 2.0*. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, 127–134. IEEE, 2010.

[8] Steve Awodey. *Category theory*. Oxford University Press, 2010.

[9] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. *Chisel: constructing hardware in a scala embedded language*. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, 1212–1221. IEEE, 2012.

[10] *SpinalHDL*. https://github.com/SpinalHDL/SpinalHDL. Accessed: 2018-08-12.

[11] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. *Cλash: Structural descriptions of synchronous hardware using haskell*. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, 714–721. IEEE, 2010.

[12] Jan Decaluwe. *MyHDL: a Python-Based Hardware Description Language*. *Linux journal*, pages 84–87, 2004.

[13] *AMBA Specification*. https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications. Accessed: 2018-08-12.

[14] *Avalon Specification*. https://www.intel.com/content/www/us/en/programmable/documentation/nik1412467993397.html. Accessed: 2018-08-12.

[15] Miguel Angel Fiol and Ernest Garriga. *Number of walks and degree powers in a graph*. *Discrete Mathematics*, 309(8):2613–2614, 2009.

[16] *Python asyncio*. https://docs.python.org/3/library/asyncio.html. Accessed: 2018-08-12.

[17] Wilson Snyder. *Verilator: Open simulation-growing up*. *DVClub Bristol*, 2013.

[18] *Moving Average Implementations*. https://github.com/damjanrak/pg_dsp/tree/a5d067/pg_dsp/moving_average. Accessed: 2019-02-11.

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io