University of Twente
EEMCS
Computer Architecture and Embedded Systems

On Productive, Low-Level Languages for Real-World FPGAs

MSc Thesis
Pieter Staal (s1776894)

Monday 28th February, 2022

Supervisors
ir. E. Molenkamp
ir. H.H. Folmer
dr. W. Kuijper
dr. ir. C.P.R. Baaij

Contents

Chapter 1

Introduction

1.1 Introduction

In this master thesis we set out to design and develop a productive, low-level FPGA design language. A productive language offers designers abstractions such that complex designs can be defined efficiently and elegantly. Low-level languages offer major control over how exactly the program text is translated to the target architecture. Consequently, in a low-level language the target architecture is in a certain way visible to the designer. To anchor this research in reality, we additionally required the language to work for a real FPGA chip.

No current language for FPGAs fulfills these criteria. The design and implementation of such a language have shown what difficulties must still be overcome to realize a production-ready productive low-level language. However, it has also shown that modern tools enable the development of such languages while this used to be impossible.

1.2 Organization

This thesis starts with a survey of computational complexities in modern FPGA synthesis flows. This research was conducted as no such overview was readily available, while having a clear view of the processes involved in synthesis and their computational complexities does provide much insight into what a low-level language must be able to do.

Our main findings are then presented in the paper *On Productive, Low-Level Languages for Real-World FPGAs*. The design of our language *Ex-PART* (*Ex*plicit *P*lace *A*nd *R*oute *T*ool – a bit of a misnomer as the final version does not explicitly allow routing constraints) Several larger hardware designs built in Ex-PART are analyzed. The results of the designs defined in Ex-PART are discussed.

As an academic paper is not a good place to present all the design decisions taken during the development of Ex-PART, an extra design decisions document is included after the main paper.

Since Ex-PART is quite a large software package its design, implementation, and usage are explained in Ex-PART manual. This manual provides a thorough explanation of every source file in the program and of every feature a designer can use. It also contains warnings for partly implemented features and behavior that one may not expect. Furthermore, a guide to setting up Ex-PART is provided.

Chapter 2

Computational Complexities in Modern FPGA Synthesis Flows

Computational Complexities in Modern FPGA Synthesis Flows

Pieter Staal
University of Twente
Enschede, the Netherlands
p.j.staal@student.utwente.nl

***Abstract*—The modern FPGA synthesis flow from hardware description to bitstream requires solving four main problems either optimally or up to certain constraints: logic synthesis, packing, placement, and routing.**

In the literature no comprehensive overview of the computational complexities of these problems is available. This paper provides such an overview. It was found that both logic synthesis and placement are NP-complete. Packing was shown to be the same problem as placement and routing in general, but polynomially solvable in special cases that occur in real FPGAs. Routing has been found to be NP-complete for a specific architecture. Optimizations in logic synthesis were surveyed: circuit minimization is Σ_2^P-complete, subcircuit substitution is NP-complete, and finite state machine extraction and optimization are efficiently solvable.

New, larger FPGAs allow FPGA engineers to build larger designs. However, such a larger design implies that the size of the problems to be solved increases as well. The found complexities indicate that run-time of tools solving these problems will grow much faster than the size of the problem does. Therefore, it is concluded that this method of defining and synthesizing FPGA designs is not sustainable.

***Index Terms*—Complexity, Digital Hardware Design, Electronic Design Automation, NP-Complete, NP-Hard, Logic Synthesis, Place and Route**

I. Introduction

A modern solution to the stagnating performance of processors is parallelism. The reconfigurable fabric of logic on an FPGA is very suitable to solve problems in parallel. In theory, this parallelism scales well: a larger FPGA immediately allows one to instantiate more of the same design, and thus solve more problem instances at the same time. Likewise, it allows larger instances of the same problem to be solved.

Larger FPGAs and larger designs imply that the problems synthesis tools need to solve also increase in size. Already it can take hours to fully synthesize a large design. With the continued growth of FPGAs, run-times will only increase.

How much run-times can increase can be predicted by examining the computational complexity of the problems involved. Formal statements and proofs on the complexity of synthesis are not available in a clear review. While it is generally known that these problems are 'hard,' it is difficult to find exactly how hard.

This paper surveys the problems a modern FPGA flow must solve to synthesize a design described in a hardware description language (HDL) such as Verilog or VHDL. For each of these problems the complexities are surveyed. Furthermore, the complexities of computing some optimizations are surveyed.

High-level synthesis is left out of this survey, as in most cases high-level languages compile to a conventional HDL, and from that point on the process is the same.

Note that, in the literature, the term *synthesis* is not just used for the process of translating a hardware description language to a bitstream. It is also used for *logic synthesis*, which is just one step of synthesis. In this paper, *synthesis* always refers to the full process.

Some background information on complexity theory necessary to understand this paper is provided here. P and NP are classes of problems. A problem belongs to class P if it is solvable by an algorithm that runs in "Polynomial time:" the run-time grows according to a polynomial function, where the variable of the polynomial represents the size of the input to the algorithm. NP is the class of problems solvable in "Non-deterministic Polynomial" time. A non-deterministic computer (which is a mathematical construct to reason about complexity) can 'guess' how to continue instead of always applying the same next computation deterministically. When solving the problem, the non-deterministic computer has the power to always 'guess' correctly allowing it to solve problems much faster. However, if a problem is only practically solvable on such a computer, it is hard to solve in real life, where such computers do not exist and must be simulated on a regular computer which has to evaluate every guess. A problem is in NP if a *polynomial time verifier* exists. This verifier is an algorithm that can determine whether some solution is indeed a solution to the problem. If it can determine this in polynomial time for any solution, the problem is in NP.

Note that if a problem is in P then it must also be in NP. We can come up with an answer in polynomial time, hence we can also verify an answer in polynomial time using the same algorithm. Whether NP is also a subset of P, and therefore P = NP, is one of the greatest open problems in computer science.

If a problem is *at least as hard* as the hardest problems in NP, this problem is said to be *NP-hard*. Note that an NP-hard problem can be much harder than some problem in NP, as NP-hardness is a lower bound.

With NP-hardness as a lower bound to complexity, it is natural to consider some upper bound. Membership of NP is exactly such an upper bound. Problems that are both NP-hard

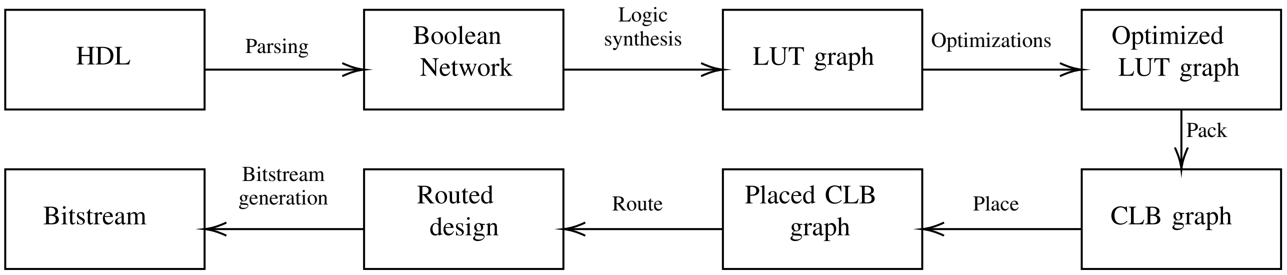

Fig. 1. A modern synthesis flow from HDL to bitstream.

and in NP are called NP-complete. Showing that a problem is NP-complete thus provides both a lower and upper bound on the hardness of the problem.

In practice, usually it turns out that if a problem is in P it is efficiently solvable, even for relatively large instances. NP-complete problems turn out to be very difficult to solve: consequently, supplying a larger input to an algorithm solving an NP-complete problem results in a much longer run-time of the algorithm. To remedy this problem a little, heuristics and approximate algorithms are used.

Complexity classes containing problems that are not in NP, but are NP-hard, also exist. One such class is Σ_2^P-complete [1]. For this paper it suffices to know that these problems are harder than NP-complete problems, but just like NP-complete problems, Σ_2^P-complete problems have a well-defined upper bound.

For more information on complexity theory, see *Computers and Intractability* [2].

II. SURVEY

A. Logic Synthesis

1) HDL conversions: Hardware is designed in a hardware description language (HDL). While Verilog and VHDL were designed with both synthesis and simulation in mind, most of the demand at the time was for a simulation language [3]. Therefore, these languages contain unsynthesizable simulation constructs. However, the synthesizable subset is still not one-to-one mappable to an FPGA. For example, processes containing if-statements do not map directly to a circuit, they must first be translated to a multiplexer tree. Therefore, HDL code must be converted to some representation that is one-to-one mappable to an FPGA.

A parsing and elaboration step converts HDL code to the intermediate representation of the synthesis tool. This problem is hard to translate to a formal statement, as it consists of many different algorithms which are applied to the parse tree and the details highly depend on the exact intermediate representation. A well-documented example of this process can be found in the open-source synthesis tool Yosys [4], [5].

2) Technology Mapping: The core problem of logic synthesis is that of technology mapping. Informally, this is the process of converting a circuit of Boolean functions to a circuit of *size K look-up tables* (K-LUTs), where K is some positive integer. Given some Boolean network B where nodes are arbitrary Boolean functions, find some Boolean network N with K-LUTs as nodes. N must compute the same function as B. Hence, a hardware description is in this step converted to a collection of combinational circuits consisting only of K-LUTs, connected via registers. It is possible to concentrate on combinational circuits since this process can be applied for every combinational path between registers, and the registers can simply be assigned as many flip-flops as their size requires.

Note that this definition for technology mapping does not specify anything about the quality of the Boolean network. If B, or the functions on the nodes of B, are sufficiently simple (i.e. have K or less inputs), the conversion can be trivially achieved by mapping every Boolean function to a size K look-up table corresponding to that function and preserving the connection structure [6]. This technique wastes look-up tables, as it does not combine functions with less than K inputs.

To make good use of the available resources it is imperative to optimize to certain resource-minimizing measures. In the case of optimization for using the least amount of LUTs, the problem is called K-RLMP [7]: find the network N consisting of K-LUTs as described above, such that the amount of nodes is minimal. Thus solving this problem minimizes LUT usage. It has been proven that this problem is NP-complete for $K \geq 3$ for combinational circuits in general [7], [8]. For other K the complexity has not been determined. Real FPGAs have look-up tables with $K \geq 3$, so solving a real-world K-RLMP instance is always NP-complete.

3) Optimizations: In synthesis tools used in industry different optimizations are applied on sub-designs, and can be applied several times [6], shrinking the inputs considerably for the latter steps, decreasing run-time and possibly delivering in a higher quality end-result.

Boolean circuit minimization optimizes the amount of gates used in combinational circuits. The problem is as follows: "given a Boolean network, find the smallest Boolean network that computes exactly the same function" [9]. Since a Boolean network corresponds to a combinational circuit, this amounts to finding a circuit that computes the same function as the input circuit using as little gates as possible. It has been shown that in general Boolean circuit minimization is Σ_2^P-complete [9]. This procedure is applied to combinational logic, so instances usually stay small, as in most hardware designs long combinational paths are cut by registers. Moreover, Boolean circuit minimization is such a common optimization problem that many heuristics have been developed. Therefore, even though the minimization problem is hard, it is possible to minimize or at least significantly shrink real-world circuits using Boolean circuit minimization. The high complexity does however imply that when instances do grow, the run-time of these algorithms increases significantly.

Subcircuit substitution is the process of identifying a part of the circuit that can be replaced by one single cell. For an FPGA that contains hardware adders, this process may identify an addition operation and synthesize it to a hardware adder instead of configuring LUTs as an adder. Ad hoc methods are able to solve this problem, for certain types of circuits. However, in general this corresponds to the subgraph isomorphism problem, which is NP-complete [10].

Finite State Machine (FSM) extraction and optimization detects finite state machines and minimizes the amount of states and control bits. This saves on routing and logic resources used by the design. For FSM extraction very efficient algorithms are available [11] which are used in practice [5]. FSM optimization entails finding the FSM with the minimum number of states performing the same function as the given FSM. Minimizing the number of states of an FSM corresponds to DFA minimization, for which an $O(n \log n)$ algorithm exists [12].

B. Packing

Modern FPGAs do not consist of variably interconnectable LUTs, but of connectable *configurable logic blocks* (CLBs), which can contain much more than just LUTs. Registers, multiplexers, adders, and DSP blocks are commonly found in such CLBs [13]. FPGAs can contain many types of CLBs, laid out in a grid and connectable via a reconfigurable routing network. Not necessarily every port of every circuit in the CLB is connected to the routing network.

Packing translates an optimized cell graph to a graph of configured CLBs. A configured CLB contains a configuration for the registers controlling its routing resources and LUTs such that it corresponds to the part of the cell graph that is mapped to the CLB. So, given a graph of cells (like LUTs, registers, adders, etc.) the packer maps these cells onto specific instances of those cells in a *type* of CLB available on the target architecture. The packer must preserve the connection structure as defined in the optimized cell graph when it packs several elements in the same CLB. The packer may optimize to use the minimal amount of CLBs for the given graph. Other optimization criteria, like minimizing delay, may be employed as well.

Notice that the packing problem, as presented here, is a place and route problem [13]. A CLB can contain both LUTs and routing resources, hence mapping some subgraph of the input graph to the internal components of a CLB is indeed a small place and route problem. This place and route problem is not just limited to the internal components of a CLB as it is not necessarily given how some part of the input graph may be divided over several CLBs.

The description of the packing problem has been kept very general. In practice, CLBs are far more homogeneous, or their structure is very simple. Since this highly restricts the problem, algorithms can solve this restricted packing problem in polynomial time: VPack [14] operates on CLBs that consist of n *basic logic elements* (BLE), partially connected to the global routing network via configurable multiplexers. VPack can pack designs for this simple architecture in $O(kC)$, where k is the maximum fan-out, and C is the amount of LUTs in the design.

Feasible solutions are obtainable in polynomial time for real-world FPGAs. However, the general case amounts to solving placement and routing.

C. Placement

The placement phase of digital hardware synthesis takes a graph produced by the packer and for each node allocates a specific CLB on the FPGA onto which that node will be placed. A good placement algorithm should place highly connected CLBs closely together to make the routing problem easier. Hence there is no one single 'Placement Problem': a measure must be defined to be optimized against and this results in problems of different complexity.

Two variations of the placement problem are presented: an optimization version and a constraint satisfaction version. The optimization version is defined as follows [15]: given are a set of modules M (which are CLB types accompanied by a configuration), positions P (these positions are physical CLBs to which a module may be mapped), signals S, a set of nets for every signal $\{N_1, ..., N_{|S|}\}$, a distance function $d(p_i, p_j)$ where $p_i, p_j \in P$, a weight function $w(N_i)$, and some cost measuring function $f(N_i, X)$ that computes the cost for a net N_i under a module assignment X. Such a function could be the minimum wire length of a net under assignment X, computed using the distance function d. With these givens, construct some assignment $X : M \rightarrow P$ such that:

1) Each module $m \in M$ must be assigned to some position $p \in P$.
2) Each position $p \in P$ has either one or zero modules assigned to it.
3) $\Sigma_{i=1}^{s} w(N_i) f(N_i, X)$ is minimal.

This definition of the placement problem corresponds directly to the Quadratic Assignment Problem [15], which has been proven to be NP-hard [16].

The other variant has the same givens and requirements, except for requirement 3) [17]:

3) $\Sigma_{i=1}^{s} w(N_i) f(N_i, X) \leq L$, for some given L.

This requirement checkable in polynomial time: simply by applying the sum and checking whether it is indeed smaller than the given L (assuming that f is a function computable in polynomial time). Therefore this version of the placement problem is both in NP and NP-hard, thus it is NP-complete.

D. Routing

In the last step of synthesis the placed CLBs need to be connected via the routing network as defined by the placed

CLB graph. Architectures of commercial FPGAs differ [18], but in general the routing network of an FPGA consists of wires, configurable switches and configurable connection points. In- and outputs of CLBs can be connected to wires via connection points and switches can be configured to connect or disconnect wires.

Because of large differences in architecture it is impractical to define one general routing problem suitable for complexity analysis [18]. A general graph-theoretic definition may be possible, but has not been attempted. A complexity result from this definition might not be useful either: routing in general may be far more complex than routing on a specific architecture.

One instance of the routing problem has been investigated in detail [19]. A Xilinx-like FPGA model was investigated where a global routing step assigns a string of connection and switching blocks to a net after which a detailed router assigns specific wires, switches and connections points for that net. It is shown that detailed routing is NP-complete [19], irrespective of the topologies of the switch blocks [20].

Little research has been done on the complexity of routing on real FPGA architectures. During most of the history of FPGAs their exact architecture was strictly proprietary and thus not available to researchers. Since any conclusions about routing depend heavily on the exact architecture hardly any results are available.

III. Conclusion

To achieve a working design from a description in an HDL it is necessary to solve at least three problems that have been partially shown to be NP-complete. Applying certain optimization strategies further increases the amount of NP-hard problems that must be solved. Finding an *optimal* solution requires solving all problems optimally at once, further increasing complexity.

To improve tractability of the synthesis problem many of the problems can be split up to smaller instances, since a smaller instance decreases run-time just as dramatically as a larger instance increases it. Further improving workability, approximations and heuristics for these problems exist.

The complexities of the problems involved show that the current method of synthesizing FPGA designs is not sustainable. It is already increasingly difficult to quickly prototype a large design with run-times as long as they are. Effectively utilizing future larger FPGAs for larger designs will take increasingly more time, severely hampering an engineer's ability to quickly prototype designs - one of the most distinguishing features of FPGAs.

References

[1] Sanjeev Arora and Boaz Barak, *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. [Online]. Available: http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264

[2] M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," 1978.

[3] P. Flake, P. Moorby, S. Golson, A. Salz, and S. Davidmann, "Verilog HDL and Its Ancestors and Descendants," *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3386337

[4] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanović, "Yosys+nextpnr: an Open Source Framework from Verilog to Bitstream for Commercial FPGAs," 2019.

[5] C. Wolf, "Yosys manual." [Online]. Available: http://www.clifford.at/yosys/files/yosys_manual.pdf

[6] J. Cong and Y. Ding, "Combinational Logic Synthesis for LUT Based Field Programmable Gate Arrays," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, no. 2, p. 145–204, Apr. 1996. [Online]. Available: https://doi.org/10.1145/233539.233540

[7] A. Farrahi and M. Sarrafzadeh, "Complexity of the lookup-table minimization problem for FPGA technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 11, pp. 1319–1332, 1994.

[8] A. Ling, D. Singh, and S. Brown, "FPGA technology mapping: a study of optimality," in *Proceedings. 42nd Design Automation Conference, 2005.*, 2005, pp. 427–432.

[9] D. Buchfuhrer and C. Umans, "The complexity of Boolean formula minimization," *Journal of Computer and System Sciences*, vol. 77, no. 1, pp. 142–153, 2011, celebrating Karp's Kyoto Prize. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000010000954

[10] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather, "*SubGemini*: Identifying Subcircuits Using a Fast Subgraph Isomorphism Algorithm," in *Proceedings of the 30th International Design Automation Conference*, ser. DAC '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 31–37. [Online]. Available: https://doi.org/10.1145/157485.164556

[11] Y. Shi, C. W. Ting, B.-H. Gwee, and Y. Ren, "A highly efficient method for extracting FSMs from flattened gate-level netlist," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, 2010, pp. 2610–2613.

[12] J. Hopcroft, "An n log n Algorithm for Minimizing States in a Finite Automaton," in *Theory of Machines and Computations*, Z. Kohavi and A. Paz, Eds. Academic Press, 1971, pp. 189–196. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124177505500221

[13] J. Luu, J. H. Anderson, and J. S. Rose, "Architecture Description and Packing for Logic Blocks with Hierarchy, Modes and Complex Interconnect," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 227–236. [Online]. Available: https://doi.org/10.1145/1950413.1950457

[14] V. Betz and J. Rose, "Cluster-based logic blocks for FPGAs: area-efficiency vs. input sharing and size," *Proceedings of CICC 97 - Custom Integrated Circuits Conference*, pp. 551–554, 1997.

[15] S. Sahni and A. Bhatt, "The Complexity of Design Automation Problems," in *Proceedings of the 17th Design Automation Conference*, ser. DAC '80. New York, NY, USA: Association for Computing Machinery, 1980, p. 402–411. [Online]. Available: https://doi.org/10.1145/800139.804562

[16] S. Sahni and T. Gonzalez, "P-Complete Approximation Problems," *J. ACM*, vol. 23, no. 3, p. 555–565, Jul. 1976. [Online]. Available: https://doi.org/10.1145/321958.321975

[17] W. E. Donath, "Complexity Theory and Design Automation," in *Proceedings of the 17th Design Automation Conference*, ser. DAC '80. New York, NY, USA: Association for Computing Machinery, 1980, p. 412–419. [Online]. Available: https://doi.org/10.1145/800139.804563

[18] Gomez, Daniel and Ciesielski, Maciej, "A Tutorial on FPGA Routing," 01 2008.

[19] Yu-Liang Wu and Tsukiyama, S. and Marek-Sadowska, M., "Graph based analysis of 2-D FPGA routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 1, pp. 33–44, 1996.

[20] Y.-L. Wu, S. Tsukiyama, and M. Marek-Sadowska, "On computational complexity of a detailed routing problem in two dimensional FPGAs," in *Proceedings of 4th Great Lakes Symposium on VLSI*, 1994, pp. 70–75.

Chapter 3

On Productive, Low-Level Languages for Real-World FPGAs

On Productive, Low-Level Languages for Real-World FPGAs

Pieter Staal
University of Twente
Enschede, the Netherlands
p.j.staal@student.utwente.nl

***Abstract*—FPGA designs are described in either Verilog or VHDL (HDLs), or in a high-level synthesis language that compiles to one of these. HDLs, however, are such an abstraction over FPGAs that the synthesis process is convoluted. Hence, hardly any *control* is available to hardware designers, hindering them in maximizing the performance of FPGAs.**

To show that languages with more control than HDLs are possible, "Ex-PART" was designed and developed: a language that enables fine-grained control over placement of modules. The hypothesis is posited that the hierarchic structure of a hardware design corresponds well with a structure of adequate placement directives for FPGAs.

Non-trivial hardware was designed in Ex-PART demonstrating that it behaves as intended. For many designs the hypothesis held true. For the largest design it did not: a much better result was obtained by placement directives that were decoupled from the program structure.

These findings show that it is possible and practical to build low-level hardware design languages in which substantial hardware can be designed. However, if designers need to optimize their FPGA designs to their maximum potential, a future low-level languages must decouple placement directives from the program structure.

***Index Terms*—FPGA, low-level language, Digital Hardware Design**

I. Introduction

The languages in use today for the description of FPGA designs were not conceived specifically for FPGAs. Verilog and VHDL (*Hardware Description Languages* or HDLs) describe digital hardware in general, not specifically targeted for an FPGA architecture. This mismatch in purposes becomes apparent with issues such as long synthesis times, and a lack of fine-grained control over the synthesis process. Contemporary HDLs do not include features that can solve these problems elegantly.

When every last bit of performance must be squeezed from the FPGA a language with a high degree of *control* is required. Compare software engineering, where an embedded engineer with very limited resources can use a language such as C to manage these in detail. A language providing much control over the resources of the target architecture is classified as a *low-level* language.

For processors, languages with a high degree of control exist: C provides all the power a software engineer needs to exert full control over the final product, while providing abstractions to efficiently develop software. These features are what is intended with a *productive, low-level language*. Such languages do not exist for FPGAs: the "lowest" level language available to hardware designers are the conventional HDLs.

For example, a feature that can increase control that is not available in HDLs is determining *where* some part of a design should be placed on an FPGA. Sometimes this is possible, but only by using synthesis directives which are not part of the language itself and may be ignored by the compiler.

In this paper, we present a new language, "Ex-PART", and a synthesis flow that implements it. The language is equipped with features allowing hardware designers to make informed design decisions on the exact location of components, or hierarchies of components, on the target FPGA.

Instead of developing an entire synthesis tool from scratch, we extended open-source tools for both logic synthesis and place and route to support our language features. This allowed us to test whether it is practical to design non-trivial hardware in Ex-PART and to evaluate the results.

In Ex-PART, a designer can specify coordinates and sizes for modules in a hardware design, facilitating fine-grained control over placement. Using coordinates and sizes for hardware modules in the language is based on the hypothesis that there is a good match between the structure of the hardware design and the structure of the placement on the FPGA.

This hypothesis is stated as follows:

Hypothesis 1 *The hierarchic structure of a hardware design corresponds well with a structure of adequate placement directives.*

However, our results show that this hypothesis does not hold in general.

The main contributions of this paper are:

1) Ex-PART, an HDL that offers fine-grained control over placement on FPGAs (Section III).
2) An extension to the open-source synthesis toolchain implementing Ex-PART (Section III-E)
3) Three hardware designs (Section IV) show that hypothesis 1 does not hold in general (Section V-A) providing important feedback on the development of other low-level FPGA languages (Section V)

II. Related Work

Research on unifying HDLs has been conducted. LLHD (Low-Level Hardware Description) [1] is a language that aims to be an intermediate representation for any hardware description language. LLHD has three versions: behavioral, structural, and netlist. Note that LLHD stops at the netlist level because LLHD needs to be synthesizable to both ASICs and FPGAs. Therefore, it is impossible to incorporate FPGA-specific information like placement and routing constraints into a language that compiles to LLHD.

A low-level FPGA language does exist: Reticle [2] aims to describe modern FPGAs at the tile level. It includes support for LUT and DSP tiles. Reticle is meant as a compilation target, and transforming Reticle to a bitstream is extremely fast because the language is so close to the FPGA hardware. As Reticle is a compilation target, it is not productive to program directly in Reticle (Similar to programming in Assembly for a processor). A language such as Reticle would, however, be an appropriate compilation target for a low-level language.

In [3] the FPGA is divided into regions in which parts of the design may be placed. The regions are then connected via a packet-switched network as a communication channel between the sub-designs. By splitting up the design, synthesis run-times can be reduced: a smaller design decreases run-time just as drastically as a larger design increases it. This strategy allows parallel synthesis of the sub-designs, further decreasing the run-time of synthesis. Designs built in this scheme should not be subdivided very often since connecting very small, simple, components via a packet-switched network wastes FPGA resources. Moreover, the strategy applied here is not integrated into a language.

III. Ex-PART

Describing hardware in Ex-PART consists of two phases: First, components are defined. Then instances of those components are laid out and connected. Inspired by C's header system, these phases happen in two separate files: respectively the "component file" and the "instantiation file".

A. Component Definition

Every component is modeled as a mealy machine, which enables the description of sequential hardware. A component without a state describes combinational hardware. A component may have any amount of inputs, states, and outputs. Each of these properties is annotated with a *type*. Inputs and outputs can only be connected if they are of the same type. States must additionally be supplied with an initial value. As an example, defining a 16-bit unsigned integer state with initial state zero and name 'counter' is written as:

```
state counter = 0 : Unsigned 16
```

The types used are those of the high-level synthesis tool Clash, which will be elaborated on in Section III-E.

For every output and state, an expression describing its relation to the inputs and previous state must be defined. The syntax for these expressions is that of a Haskell expression. The current state and next state are differentiated by adding a prime (′) to the state name to designate the next state.

For example, incrementing the state 'counter' by some amount set on the 'interval' input is denoted as:

```
counter′ = counter + interval
```

$\texttt{third}^{1}\ \texttt{is compute}^{2}\ \texttt{in (5, 5)}^{3}\ \texttt{at (second.x}^{5}\texttt{ + second.w,}^{6}\texttt{ 0)}^{4}$

Fig. 1. Example instantiation of a component. 1. Name given to this instance; 2. Name of component to be instantiated here; 3. Width and height of component; 4. Coordinates of the top left corner of this component; 5. X property of instance named 'second'; 6. Width property of component named 'second'.

$\texttt{second}^{1}\texttt{.sum}^{2}\ \texttt{->}^{3}\ \texttt{third}^{4}\texttt{.part}^{5}$

Fig. 2. Example of a connection between the output port of one component instantiation and the input port of another; 1. Name of the source instance; 2. Name of the output port of this instance; 3: Arrow syntax to denote connection; 4. Name of the destination instance; 5. Name of an input port of the destination instance.

The complete code of a simple combinational component is shown in Figure 3.

It is possible to add arbitrary Haskell code in a `haskell` block. In such a block, data types and type synonyms may be defined as well. These definitions can be used in every component definition.

B. Instantiation and Connection

The components defined in the component file are laid out and connected in the instantiation file. Instantiation files are structured as a hierarchy of *modules*. Modules can have inputs and outputs annotated by types as in component definitions. A module is located at a certain position on the FPGA and is of a certain size. These properties are given through x and y coordinates and width and height numbers. In denoting these properties, the size and location information of other modules and components may be used. Modules may contain submodules, component instantiations, and connections. All coordinates used in one module are relative to the top-left corner of that module. Submodules define a hierarchy in modules and ease working with the coordinate notation. Component instantiation statements place a component defined in the component file at a certain position and allocate some rectangular area for the component, as shown in Figure 1. Again, this position is relative to the position of the module in which this component is instantiated. Connections connect ports from components and modules to other ports. An example of connecting the output port of one component to the input port of another is given in Figure 2.

Note that this definition allows overlapping components. Overlap gives the placer more freedom to find better placements. Good placements on FPGAs are often not rectangular, or components may be interconnected thoroughly. In these cases some or full overlap of components yields much better results. Hence, the hardware designer can decide how large the solution space is that the placer must explore by adjusting the amount of overlap.

To ease the development of larger designs, features are available for repetitive layouts of components and their connections. These language constructs are all elaborated to modules utilizing only submodules, instances, and connec-

```
component on_odd() {
   input value : Maybe Value
   output result : Maybe Value

   result = case value of
       Just v -> Just (v * 3 + 1)
       Nothing -> Nothing
}
```

Fig. 3. Description of the component `on_odd`

tions. Documentation and usage examples are available in the repository [4].

C. Design Feedback

Overviews of metrics like area usage, maximum frequency, and synthesis times for the entire design and for individual components can be provided by a script.

Both the bitstream and the location information can be visualized. The visualizer shows changes immediately when compiling. Therefore, it is possible to monitor important properties of the design during development.

Clash can be used to verify the functional correctness of a design. It provides an interactive interface that allows simulating the defined module and all its submodules with arbitrary inputs. It is possible to automate these tests with testing libraries.

D. Example Design

This walk-through exhibits the design process of a simple module in Ex-PART. This module applies the rules of the Collatz conjecture to an integer n in a register. After cycle i, the value of n is given by:

$$n_{i+1} = \begin{cases} n_i/2 & \text{if } n_i \text{ is even} \\ 3 \cdot n_i + 1 & \text{if } n_i \text{ is odd} \end{cases} \tag{1}$$

Components: To realize this design five components are defined in the component file.

1) `control` stores the value of n, and sets a new value for n if it is provided on its input.
2) `on_odd` performs the computation for odd n.
3) `on_even` performs the computation for even n.
4) `router` sends n from the controller to either `on_odd` or `on_even`, depending on the parity of n.
5) `merger` takes the result from either `on_odd` or `on_even` and sends it to the `controller`.

The components are to be connected as shown in Figure 4, where the arrows are annotated with the type of the ports they connect.

The definition of the `on_odd` component is shown in Figure 3. Both its input and output are of type `Maybe Value`: there can either be a value on the input, or `Nothing`, signifying that there is no input. In hardware, this would be translated into a port with enough bits to contain the representation of `Value`, and one extra bit signifying whether the value is `Nothing`. The relation between the input `value` and the output `result` is defined in the case expression. It states that if `value` is of the form `Nothing`, emit `Nothing` again. If it does contain

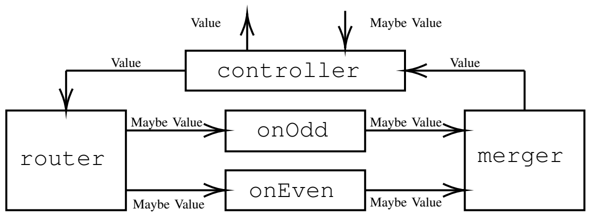

Fig. 4. Architecture for the Collatz conjecture calculator.

a value, multiply that value by three and add one, as is the definition for the odd case in equation 1. Notice that this is a combinational piece of hardware: the relation between input and output is defined directly, without state.

When a component has been defined in Ex-PART, one can inspect properties such as area usage. The visualizer can render these results and an analysis script gives a quick overview of resource usages reported by the tools.

Using similar information for the remaining components of the Collatz design, a layout is defined in the instantiation file which is shown in Figure 5. Plenty of space is taken for each component and connections are not shown. An explanation of important lines is provided below.

1: The definition of the top-level module. The top-level module must have a constant expression for its coordinates, as its position is what the coordinates of every submodule and component depend on. It is instantiated at (2, 2) as on the target architecture the two leftmost columns and the two topmost rows do not contain any LUTs. An area of six by five tiles is deemed sufficient for the whole module.

2: Modules have inputs and outputs. As these are the I/O of the top-level module, they will be connected to the I/O pins of the FPGA.

5: This is one of the component instantiations. In the component file, the `control` component was defined. This instantiation has been given the name `controller` and an area of six tiles at the top of the module is allocated.

7: The submodule `update_n` is defined. Its location is relative to its parent module's location, thus we can instantiate it at `(0, controller.h)` to place the whole module exactly under the controller component. Its width is set equal to that of the controller, and height is set as the remaining module height.

8: Similarly to the top-level module, the `update_n` module also exposes its internal data to two ports. An input and output port are defined. This definition shows that we can place a value at the input of this module, and the module will return the new value according to the rules of the Collatz conjecture.

13: The instantiation of the `on_odd` component. Its width and location are defined in terms of other components to make resizing and moving easier.

Connections: Every component has input and output ports. These ports can be referenced in a connection statement. The

```
collatz in (6, 5) at (2, 2) {
    input setting : Maybe Value
    output result : Value

    controller is control in (6, 1) at (0, 0)

    update_n in (controller.w, 4) at (0, controller.h) {
        input value_in : Value
        output value_out : Value

        router is router in (1, on_odd.h + on_even.h)
           at (0, 0)
        on_odd is on_odd in (update_n.w - 2, 2)
           at (update_n.x + 1, 0)
        on_even is on_even in (on_odd.w, on_odd.h)
           at (on_odd.x, on_odd.h)
        merger is merger in (1, on_odd.h + on_even.h)
           at (on_odd.x + on_odd.w, 0)
    }
}
```
Fig. 5. Instantiation file for the Collatz example.

connections in the `update_n` module are shown in Figure 6. An explanation is provided of the important lines:

1: When no component is specified, as with `value_in` in this statement, the current module's I/O ports are intended. Hence, this connection statement connects the input `value_in` of the `update_n` module to the input `value` of `router`. As these are both of the type `Value`, this connection is possible.

2: This is a connection internal to the `update_n` module. The `odd` port of the `router` component is connected to the `value` port of the `on_odd` component.

4: The `result` port of `on_odd` is connected to the `value_odd` port of `merger`.

6: The value at the `result` port of `merger` is the result of the calculation and should therefore be routed out of the module via the `value_out` port.

The rest of the connections can be found in the complete example in the repository [4].

Visualization: Since coordinates usually do not speak to the imagination, we also supply a visualizer that can show both bitstreams and layout definitions. Therefore, it is possible to check with the visualizer whether the layout is as intended. The layout defined in Figure 5 is shown in Figure 7.

Synthesis: With the components, layout, and connections specified the design can be synthesized and the result is shown in Figure 8. This is a visualization of the bitstream generated by the tools Ex-PART employs. This bitstream can be flashed to a Lattice ECP5 FPGA.

This visualization should be read as follows: FPGAs consist of a grid of *tiles*. Every square in the visualization is such a tile. Each tile on the ECP5 contains four *slices*, which are drawn as a colored horizontal rectangle. A slice contains two registers and two 4-bit LUTs. Notice that the background of

```
1 value_in->router.value
2 router.odd->on_odd.value
3 router.even->on_even.value
4 on_odd.result->merger.value_odd
5 on_even.result->merger.value_even
6 merger.result->value_out
```
Fig. 6. Connections defined in the `update_n` module.

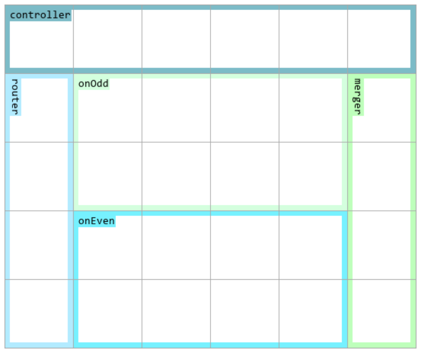

Fig. 7. Layout of the Collatz example, as shown by Ex-PART's visualizer.

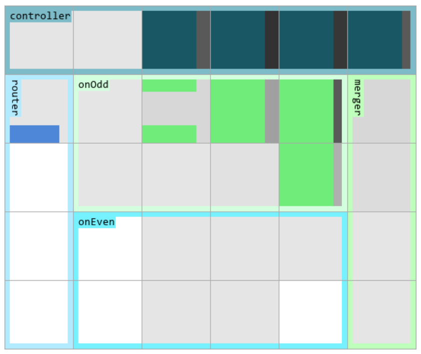

Fig. 8. visualization of the bitstream as synthesized for the Collatz conjecture example.

a tile is not always white: the brightness of a tile illustrates the usage of routing resources on that tile: the darker it is, the more resources are used.

Some conclusions can be drawn based on this visualization. Conceptually, `on_odd` and `on_even` perform similar tasks. In hardware these differ: dividing by two is the same as a shift of one bit which is just a rearrangement of wires while calculating the odd branch requires LUTs. Therefore, fewer resources should have been allocated to the `on_even` component. Similarly, the router uses just one slice, but four tiles have been allocated. On the other hand, examining the `controller` shows that decreasing its area may not work: it uses two-thirds of its slices and many routing resources. With this information, the design can be reworked to use much less area. Thus the visualization gives valuable feedback on the resource usage of components.

E. Implementation

To be able to focus on language design, and target a real FPGA, existing tools were repurposed. This choice resulted in a compilation process that seems quite elaborate; however, it actually enabled rapid prototype development.

Auxiliary Tools: To synthesize Ex-PART we use three existing tools for Verilog generation, logic synthesis, placement, and routing. All tools used by Ex-PART are open-source. Using open-source tools as the back-bone of the language gives us the freedom to inspect the processes employed whenever necessary. This, along with well-document input and output formats, allow fast development of experimental software extending these tools.

Clash [5] is used to generate Verilog. Clash is a high-level synthesis tool that can transform hardware descriptions in Haskell to Verilog or VHDL. Haskell's syntax was deemed very concise and appropriate for the description of expressions in Ex-PART, and it is relatively easy to generate programmatically. By repurposing Haskell's syntax in Ex-PART it is straightforward to generate Clash components from a description in Ex-PART.

Yosys [6] is an open-source logic synthesis tool. It can synthesize Verilog for some FPGAs. Yosys' outputs are in the JSON format which is human-readable and easy to work with programmatically. The specification of the output is well-documented. For features absent in the documentation, Yosys' code is available for inspection.

To perform placement and routing for the ECP5 FPGA *nextpnr* [6] is used. This tool can perform placement and routing given a JSON file generated by Yosys. It can output a JSON file describing the placed and routed design, or a textual configuration file that can easily be converted to a bitstream that can be flashed to the ECP5.

Compilation: In summary, compilation from Ex-PART to the ECP5 FPGA consists of the following steps:

1) Parse the component and instantiation file.
2) Elaborate the design.
3) Calculate absolute positions.
4) Generate Clash for simulation and synthesis.
5) Compile Clash to Verilog *per component*.
6) Synthesize the components with Yosys.
7) Connect and instantiate components as defined in the instantiation file.
8) Place and route the resulting design with nextpnr, constraining locations.

The component and instantiation files are read and parsed. The parsing process produces data structures representing the hardware design. Many of the constructs available in Ex-PART can be elaborated into simpler constructs. Thus elaboration converts the design into a design that only uses connections, component instantiations, and submodule definitions. The rest of the compilation is applied to the elaborated data structure.

A file (`locations.json`) with absolute coordinates of the top-left and bottom-right corner of every component instantiation is generated. This file is used in constraining LUT locations during placement.

Every component is compiled to its own Clash file. Such files can be loaded in the interactive Clash environment to validate the functional correctness of the design through simulation.

For every component, the generated Clash is compiled to Verilog, as that is the format accepted by the synthesis tool Yosys. Notice that synthesis has to be applied only once for every component, even if it is re-used. The entire design is *not* compiled to Verilog: only every bottom-level component that is defined in the component file.

The generated Verilog files are synthesized with Yosys. Yosys produces a JSON file describing the design in terms of look-up tables and flip-flops. Note that Yosys could be configured to synthesize to more complicated cells, such as DSP blocks. This does imply that when a component uses such cells, the designer must place the component in an area that contains such cells. After logic synthesis by Yosys, the connections, instantiations, and hierarchy described in the instantiation file are converted into Yosys' JSON format and appended to this file.

The resulting JSON file is processed by nextpnr to place and route the design on the ECP5 FPGA. A Python script is executed just before the placement step to constrain the placement of look-up tables. The script looks up the rectangular area a LUT should be in, in the `locations.json` file that was generated earlier. If the designer did not supply enough area for a component, nextpnr will throw an error.

If every component did indeed get enough space for all its LUTs, registers, and connections, nextpnr produces textual bitstream files, which can be converted into a binary bitstream for programming an ECP5 FPGA. The visualizer can show where all the LUTs are placed.

Note that in this implementation *constraints* are added to the placer which may make the problem at hand harder to solve. We chose this strategy as it was the fastest route to realization.

A detailed description of the compilation process and the complete code are available in the repository [4].

IV. Results

A. Compilation Flows

Ex-PART comes with three alternative compilation flows to compare the results of different synthesis strategies.

1) *Ex-PART* takes location and size annotations into account when synthesizing the design.
2) *Hierarchic*: The design is split into Verilog modules as a designer using a conventional HDL would do. Every component is compiled to a separate Verilog module. This generates HDL code organized similarly to how a hardware designer would have. No placement constraints are applied.
3) *Monolithic*: No modules are generated at all. One fully flattened design is generated such that the synthesis tool can optimize between module interfaces.

For three examples the placements are shown: the Collatz example from Section III-D, a parallel MD5 hash calculator, and a simple manycore. To further compare the costs and benefits of each flow, performance metrics like maximum frequency, area usage, and synthesis run-times for all three flows are given.

B. Collatz Conjecture Calculator

The placement found for the Collatz conjecture calculator is shown in Figure 8. The results of running all three flows for this module are shown in Table I.

	Ex-PART	Hierarchic	Monolithic
LUTs	54	54	49
f_{max} (MHz)	122.29	132.01	148.46
Flip-flops	32	32	32
Synthesis time (s)	0.75	0.74	0.6
Placing time (s)	0.05	0.04	0.04
Routing time (s)	0.07	0.06	0.1
Total time (s)	0.87	0.84	0.74

TABLE I
RESULTS FOR THE COLLATZ CONJECTURE CALCULATOR.

C. MD5 Hasher

MD5 [7] is a hashing algorithm. Although it has fallen out of use for cryptographic purposes, it is a good demonstrator for Ex-PART as the calculation of an MD5 hash is not trivial.

This design can hash 128-bit messages. These messages are provided in 32-bit chunks. When four chunks have arrived, the hasher starts. It applies the MD5 algorithm in 64 cycles. When finished, the 128-bit hash is emitted in 32-bit chunks in four cycles.

Module re-use is used to instantiate the hasher four times, demonstrating Ex-PART's productivity. A load balancer divides the incoming message words over the four hashers to compute four hashes in parallel.

The placement found by the hierarchic flow is shown in Figure 9. The default placement algorithm found a rectangular area and a roughly symmetric placement for the four hashers. The total design takes up a rectangle of 44 by 29 tiles, or 1279 tiles in total. For the sake of showing that Ex-PART can impart control over placement, suppose instead a linear layout is desired. Such a layout was defined in Ex-PART, and the result is shown in Figure 10. This layout fits in an area of 20 by 62 tiles, occupying 1240 tiles in total. Hence, this layout uses a very similar amount of space as the hierarchic flow, showing that a linear layout could be achieved without wasting area.

In Table II the rest of the metrics are shown. The monolithic run takes much more time to synthesize as it applies logic synthesis to the entire design at once, instead of optimizing smaller modules separately. The place and route time of the Ex-PART flow is longer than the hierarchical. This is due to the stricter constraints on placement that are applied by the designer in Ex-PART. The hierarchical design is free of such constraints. Note that the quality of the designs does not differ much: the maximum frequency f_{max} is nearly identical for every flow.

	Ex-PART	Hierarchic	Monolithic
LUTs	6606	6602	6249
f_{max} (MHz)	17.04	17.45	18.38
Flip-flops	3176	3176	3176
Synthesis time (s)	5.02	5.91	25.32
Placing time (s)	49.56	31	30.38
Routing time (s)	25.76	12.43	12
Total time (s)	80.34	49.34	67.7

TABLE II
RESULTS FOR THE FOUR MD5 HASHERS.

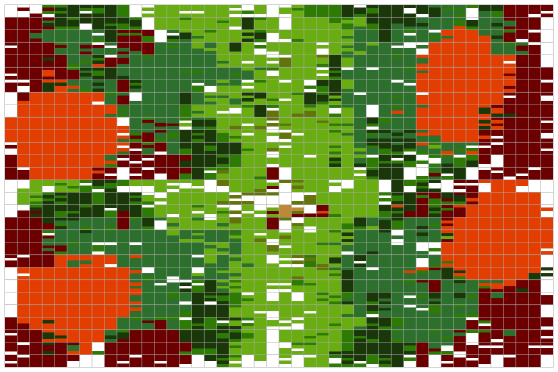
Fig. 9. Placement of four MD5 hashers by the hierarchic flow.

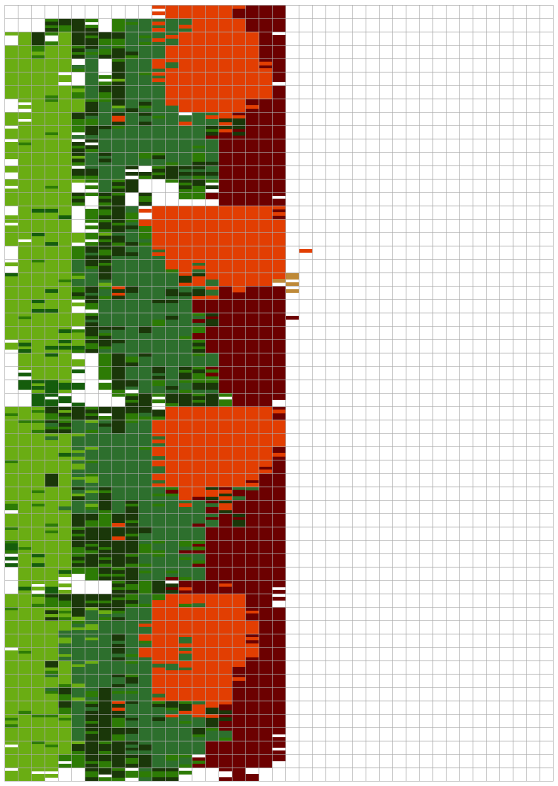
Fig. 10. Linear placement of the MD5 example, constrained as such by the placement directives in Ex-PART.

D. 4×4 Manycore

A manycore was designed to exhibit designing hardware requiring more resources in Ex-PART. The manycore consists of very elementary processors with four 8-bit registers, six instructions, and a program memory of 32 instructions. They are connected via a packet-switched network in a grid topology. Each processor is paired with a router providing access to the on-chip network. This Processing and Routing Unit (PRU) is laid out in a grid. The processor and router were kept simple to be able to synthesize many PRUs for a relatively small FPGA.

The architecture of a PRU is shown in Figure 11. The processor reads incoming packets from a FIFO and pushes packets into the routing network via another FIFO. The router

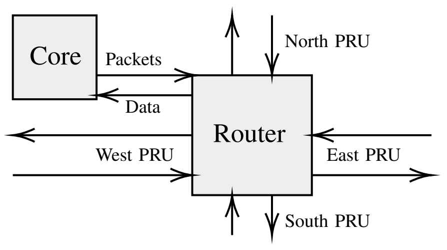

Fig. 11. Design of a Processing and Routing Unit, the main building block of the manycore.

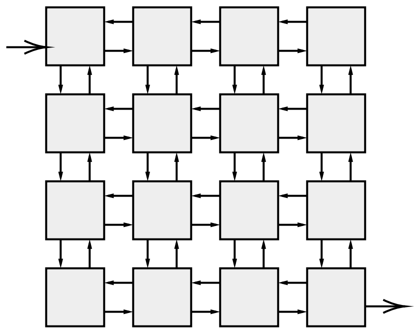

Fig. 12. Architecture of the 4 × 4 manycore.

uses a simple decision procedure based on its location and the destination location in the packet to send incoming packets to their destination.

The architecture of the manycore is shown in Figure 12. The cores are linked to their neighboring PRUs through the routers via their north, east, south, and west I/O ports. On the fringes of the network one input and one output are located to feed data into the network and receive results from the cores. These are constrained to I/O ports of the FPGA.

A space-efficient placement of the manycore was hard to achieve. In Figure 13 one placement obtained by Ex-PART is shown, and in Table III the metrics of synthesizing this design are shown. The Ex-PART flow is faster than the hierarchical flow because the hierarchical flow places the cores close to eachother while in Ex-PART the designer can pre-emptively leave room between the PRUs to make routing easier.

Notice the discrepancy between the amounts of LUTs used between the three flows. The monolithic flow manages to optimize out one-third of the LUTs, resulting in saving a huge amount of time during placement and routing and obtaining a higher maximum frequency. The main difference between the Ex-PART and hierarchic flow, and the monolithic flow is that the monolithic flow can optimize over the interfaces between modules. This strategy increases synthesis run-time, as is clear from the results: the monolithic flow runs more than ten times as long. In this case, however, the extra time spent during synthesis saves much time during placement and routing.

Upon further inspection, the large optimization was achieved mainly between routers. The conceptual design of this manycore favors routers and processors to be grouped as one unit, the PRU. However, this conceptual separation of routers in the network implies a physical separation as well, by hypothesis 1. When the design was arranged such that the entire routing network was one optimizable module, indeed a similar result as the monolithic flow was found.

	Ex-PART	Hierarchic	Monolithic
LUTs	14560	14848	9954
f_{max} (MHz)	58.3	57.4	74.24
Flip-flops	17632	17632	14932
Synthesis time (s)	3.93	7	87.2
Placing time (s)	128.63	147.26	63.06
Routing time (s)	353.42	2007.7	35.78
Total (s)	485.98	2161.96	186.04

TABLE III
RESULTS FOR THE MANYCORE.

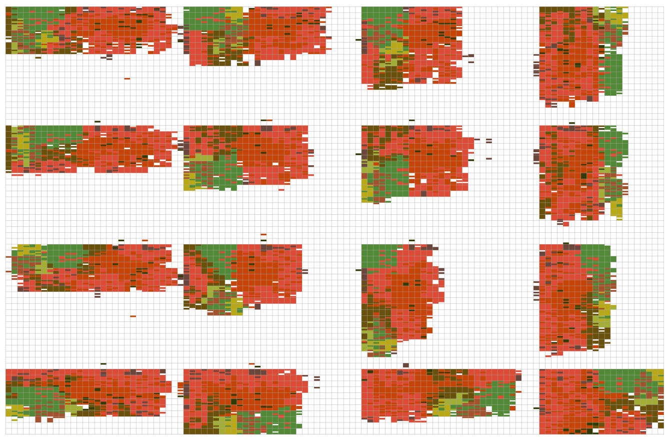

Fig. 13. A placement of the 4 × 4 manycore found through Ex-PART. Each PRU is clearly visible as a separate entity.

V. DISCUSSION

A. Hypothesis

During the design of Ex-PART, hypothesis 1 on the correspondence of design hierarchies and adequate placements was assumed to hold. The results of the MD5 design show that there are designs for which the hypothesis is true; however, the development of the manycore shows that the hypothesis does not hold in general: the structure of a design may be pleasant to work with, easy to expand, and easily modified, but using this structure as a directive for placement did not yield an adequate placement. Hence, it is critical for a low-level language for FPGAs to take into account that the structure of the design cannot be linked in a one-to-one fashion to directives involving low-level features. This result is vital for the further development of Ex-PART, or the design of other low-level languages.

The manycore design did show that the designer had control over placement. In this case, the design for the routing network was split up such that certain optimizations were not detectable for the tools; hence, the conceptual structure of the design induced a sub-optimal design. However, in the placement the conceptual separation of the PRUs *is* still visible, showing that Ex-PART works as intended.

B. Open-source Synthesis Tools

To gain more control over synthesis, placement, and routing of hardware designs tools must provide clearly documented input and output formats: the fact that Yosys and nextpnr do provide this, entirely enabled the development of Ex-PART. If every step in a synthesis flow produces and consumes open files anyone can inject their own processing just as has been done for Ex-PART enabling the development of low-level FPGA languages and thereby giving hardware designers just as much freedom in their tools as software engineers.

When the tools are open-source as well, any lacking documentation can be compensated for by investigating the source code. Bugs in custom processing can also be resolved more conveniently when the source code is available for debugging.

C. Routing

Ex-PART focuses on placement; however, a low-level language could allow hardware designers to put constraints on routing as well. The granularity of this control can vary immensely: a language may require designers to specify paths completely, or allow designers to specify parts of paths while the tooling figures out the rest. The design of a language that effectively incorporates this control in its syntax and semantics is future work.

D. Synthesis Speed

To apply the placement directives, Ex-PART supplies extra constraints to the place and route tool. This method may introduce harder to solve constraints resulting in longer place and route times. However, it did enable the fast development of Ex-PART as this method is supported by nextpnr. A low-level FPGA language could be designed with synthesis speed in mind. A language where hierarchies are synthesized, placed, and routed completely separately can be much faster than current synthesis tools. Synthesis consists of three NP-complete or harder problems [8], thus having the designer split up the design into smaller problems decreases the individual run-times of the sub-designs such that the sum of their run-times is smaller than synthesizing the entire design at once. The smaller designs must then be connected correctly again, this corresponds to another (quite small!) routing problem. Such strategies are only possible when the designer annotates where the design can be split up.

E. Dynamic Partial Reconfiguration

Language features constraining placement as in Ex-PART may be combined with dynamic partial reconfiguration. Defining rectangles for modules within the language in which the hardware is designed explicitly shows when components can be interchanged. A tool strictly enforcing such area allocations eases working on reconfigurable designs.

VI. Conclusion

The development of Ex-PART was sparked by the lack of languages offering fine-grained control over synthesis, placement, and routing of hardware designs intended for FPGAs. In the design of Ex-PART, the assumption was made that the structure of a design corresponds well with a structure of adequate placement directives.

To evaluate Ex-PART three designs were developed: a Collatz conjecture calculator, a parallel MD5 hash calculator, and a rudimentary manycore. For the first two designs, Ex-PART proved very effective: the designs could be laid out in a largely different layout from the regular synthesis tools, proving that Ex-PART grants designers more control.

However, the manycore design demonstrated that linking placement directives to design structures does not always yield good results. The conceptual separation of cores in the manycore worked well during development, but inhibited the tools from making an extreme optimization.

This shows that the hypothesis does not hold in general. This result is critical in the development of other low-level FPGA languages as it implies that directives increasing control should to be decoupled from hierarchic design constructs.

Ex-PART opens up a lower level FPGA programming style previously inaccessible to hardware designers. With placement design decisions embedded inside the language, and tools available to make these decisions in an informed manner, placement can be performed *while* describing the hardware. It has been shown that developing a language like Ex-PART is highly feasible with modern open-source synthesis and place and route tools. New productive, low-level languages will open up FPGAs enabling skilled hardware designers to use FPGAs up to their maximum potential.

References

[1] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, “LLHD: A multi-level intermediate representation for hardware description languages,” *CoRR*, vol. abs/2004.03494, 2020. [Online]. Available: https://arxiv.org/abs/2004.03494

[2] L. Vega, J. McMahan, A. Sampson, D. Grossman, and L. Ceze, “Reticle: a virtual machine for programming modern FPGAs,” in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 756–771. [Online]. Available: https://doi.org/10.1145/3453483.3454075

[3] Y. Xiao, D. Park, A. Butt, H. Giesen, Z. Han, R. Ding, N. Magnezi, R. Rubin, and A. DeHon, “Reducing FPGA compile time with separate compilation for FPGA building blocks,” in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 153–161.

[4] P. J. Staal and H. H. Folmer, “Ex-PART.” [Online]. Available: https://github.com/PietPtr/Ex-PART

[5] “Clash,” https://clash-lang.org, accessed: 2021-12-21.

[6] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovi, “Yosys+nextpnr: an open source framework from Verilog to bitstream for commercial FPGAs,” 2019.

[7] R. Rivest, “The MD5 message-digest algorithm,” Internet Requests for Comments, RFC Editor, RFC 1321, 4 1992. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1321.txt

[8] P. J. Staal, H. H. Folmer, “Computational complexities in modern FPGA synthesis flows,” *unpublished*, 2022.

Chapter 4

Design Decisions

4.1 FPGA Design Descriptions

The lowest level hardware description languages available to FPGA engineers are Verilog and VHDL. The results from the survey paper on computational complexities [1] show that the conversion from HDLs to an FPGA design is three times NP-hard. Furthermore, HDLs obscure many features inherent to FPGAs: where some part of the design is placed and how it is routed is entirely hidden from the perspective of an HDL. Hence, HDLs are not very low-level at all.

To examine what kind of language could be a good low-level language for an FPGA, we decided to start with examining the architecture of a specific FPGA (the Lattice ECP5). We took the ECP5 FPGA and forgot about every programming toolchain and from that perspective we designed a hypothetical toolchain from the ground up, balancing abstracting away cumbersome details of the FPGA architecture with preserving transparency.

The first step in the design of this hypothetical toolchain was an assembly-like language for the ECP5. In this language every LUT and register could be configured in as much detail as was available on the ECP5. This description of a design on an FPGA can be trivially compiled to a bitstream. This idea is indeed not new, Reticle [2] is such a language where the designer must specify the exact location and configuration of a tile on the FPGA. Their compilation speed is of course lightning fast: they make the designer specify so many details that no NP-hard problems need to be solved at all.

Building a large design in such a language is not productive as specifying the configuration and location of every tile by hand is a huge task. To obtain a productive language, language we started abstracting away details from that low-level language to provide productivity while maintaining transparency. This process of abstraction is what led to the current design of Ex-PART. These abstractions were made with attention to computational complexity of synthesis in mind, however this was a secondary goal. Sadly, in the current implementation of Ex-PART not much gains were made in synthesis speed in general. However, as discussed in the paper, Ex-PART's syntax and organization do open up avenues to synthesis strategies that may significantly speed up synthesis. See also Section 4.8 of this document.

4.2 Design Representation

Hardware description languages represent designs differently. Ex-PART must use a representation that is easily annotated with low-level directives. First we introduce how traditional HDLs and Clash represent their designs.

In Verilog and VHDL, a hierarchy of *modules* represents the hardware design. These modules define their inputs and outputs, and state is defined as registers belonging to that module. Inside the module combinational and sequential constructs can be applied to the input signals and state to define the output signals. Each module may instantiate other modules to create a hierarchy.

In Clash one can lift pure Haskell functions to the domain of signals by using the functions `moore` and `mealy` to generate a Moore or Mealy machine from a specific type signature, or one can utilize the applicative functor instance of the `Signal` datatype to apply pure functions over signals. These structures allow powerful functional programming patterns to be used to describe hardware designs on many levels of abstraction.

For Ex-PART a limited version of a module representation is used. We view the hierarchy defining a hardware design as a tree of module instantiations. The leaves of this tree are all Mealy machines, and the nodes are groups of instantiations of predefined Mealy machines, or definitions of other submodules. This design is more limited than Verilog and VHDL as it only allows logic to be defined on the leaves of the tree, not in the nodes where only instantiations and connections between modules can be defined. This design representation also does not allow any of the powerful programming patterns Clash features. What this design *does* allow is clear places for placement annotations: the nodes and leaves of the tree. Furthermore, tree structures are easy to work with and implement, and enable such features as placement relative to parents.

Restricting logic to the leaves made implementation easier. Furthermore, this logic could have been represented differently, e.g. by a Moore machine or a set of processes. Mealy machines are used as they support both combinational and sequential hardware: To build a combinational circuit, leave out the state. To define memory elements, simply drive the value of the state unmodified to the output. Furthermore, it is easy to generate the definition of a mealy machine in Clash programmatically, making implementation of Ex-PART easier.

4.3 Separate Component Definitions and Instantiations

Designing hardware in an HDL consists of defining a hierarchy of modules, each operating on their inputs and state to produce an output. When that definition is ready, the design is synthesized. If synthesis succeeds in reasonable time, the design can be uploaded to an FPGA.

In Ex-PART, more control is given to the designer. With this control the designer can improve the chances that synthesis succeeds. This does add an extra phase in the design of hardware in Ex-PART. In the first phase of development the designer defines the modules themselves in the form of Mealy machines, this is quite similar to development in a traditional HDL. In the second phase the hierarchy and connections between those modules is defined and the components and hierarchies are laid out physically on the FPGA.

To explicitly show that these phases are separate concerns, they are performed in separate files. Component definitions are written in `.expc` (`c` for component) files, and hierarchies of instantiations in the `.expi` (`i` for instantiation) file.

4.4 Layout Expressions

The goal of Ex-PART is to allow a lower level of hardware descriptions, and specifically to provide language constructs to influence placement.

Some other work has been done in designing (embedded) languages with language constructs for placement. Wired [3] uses a relative system: components are laid out next to each other. Thereby allowing descriptions along the lines of "*component a is below component b, component c is to the left of component a*". Reticle [2] on the other hand uses a coordinate system. Elements of the system can be placed by the designer on an (x, y) coordinate.

To denote positions and sizes we chose to use a coordinate-based system, i.e. positions are denoted by an x and y value and sizes by a width and height value. With this system it is completely clear where a cell will be placed. Furthermore, this matches well with the tools as they also use a coordinate system for tiles on the ECP5 FPGA. Since coordinates are not easy to reason about for humans a visualizer is available. Having the visualizer in the design loop helps the designer verify their coordinate expressions. Another measure that makes defining placements easier is the fact that coordinates need not be expressed absolutely but can be denoted as an expression whose terms are other coordinates and sizes.

4.5 Component Shapes

In Ex-PART the designer is able to place a component in a certain area. The ideal shape of this area differs per component and it may also change when the component is connected to another component. For example, a component on its own may be best synthesized in a rectangular fashion, while when it is synthesized as a sub-component of a complicated hierarchy it may take on a more circular shape.

Synthesis tools usually generate standalone designs in a rectangular pattern, showing that any other shape usually is not necessary. Furthermore, we need to take into account that defining a shape in a text-based format is not human friendly. Allowing the definition of any shape would lead to much more syntactical noise causing confusion. Experience in designing in Ex-PART shows that a visualizer is necessary to efficiently work with defining rectangles using coordinates and sizes. These factors motivate the choice of limiting Ex-PART component shapes to rectangles.

Ex-PART allows overlap in component shapes. The exact consequences of overlapping components varies wildly per component and by the degree of overlap. To illustrate some of the consequences one may encounter we elaborate two examples.

Two components may not contain much logic but are connected via a high bit width connection. If one does not define areas for these two components that overlap, all the connections must go between the edge of the two areas that they share. This results in a complicated routing problem. To make the routing problem easier these components can be given a partly or fully overlapping area to be synthesized in. Now the placer can place the cells of both components in an interspersed fashion such that the routing problem becomes easier again.

It may be nice to define a component in terms of very small components, but the degree of control intended may be more coarse grained. Take for example the manycore described in the paper in Chapter 3, where a PRU was treated as one component during placement: all the sub-components of any given PRU were given the same shape to be placed in. This is done as every PRU needs a slightly different placement of its sub components: the top-left most PRU benefits from having its routing logic placed in the bottom right, nearer to the other PRU's. For the bottom-right PRU the reverse holds. Giving every sub component of a PRU the same area gives the placer the chance to find this out on its own. As the PRU design is quite small, the placer can detect this in little enough time. This experience with the manycore again confirms the conclusion in the paper to decouple program structure from low-level directives. The program structure of a PRU consists of many tiny components. To describe the entire manycore these PRU modules can simply be repeated into a 2D grid. To place these efficiently every PRU needs a different placement, which in Ex-PART requires the PRU's instantiations *and* connections to be completely redefined. This is not an ergonomic way of programming a design.

4.6 Chain and Repeat

A goal of Ex-PART was to allow *productive* low-level development. Hence, it must be possible to abstract over designs in a certain way. For inspiration on a solution for this we looked at Clash. In Clash, components can easily be instantiated often by applying `map` and `fold` functions. We decided to use slightly more specialized versions of these functions and to rename them such that they make more sense in a physical context. Chain and repeat are specialized versions of map and fold because chain and repeat only describe a layout, while map and fold are higher order functions that *in the context of a hardware design* describe a layout. Hence map and fold are much more general as in other contexts they describe different procedures while chain and repeat are simply layout and connection descriptions.

Since `map` applies a function to each element in a vector it is analogous to instantiating that function once for every component, i.e. the function is *repeated* for every element. Thus, with the keyword `repeat` we can repeat components in Ex-PART and provide placement constraints for this repetition of components.

`fold` not only applies a function to every element, this function also takes the result of the function operating on the previous element. This results in a *chain* of function instantiations. In Ex-PART the `chain` keyword instantiates such a chain of functions.

Experience in programming in Ex-PART was gained when developing the example designs for the paper. We will elaborate a little on these (hard to quantify, slightly subjective) experiences regarding chains and repeats here. As mentioned, chain and repeat statements in Ex-PART are a bit more specialized than maps and folds in Haskell. Working with chains and repeats showed that their expressiveness was a little more limited than expected: the possibilities for abstraction that Haskell offers are much greater than Ex-PART, and some designs could have been written in a more elegant or terse fashion with those abstractions. Programming with Ex-PART's chain and repeats also immediately gives insight into what structure is being described, while in a VHDL for-loop that is only clear after careful inspection of the body. Exact statements on the power of the chain and repeat abstraction relative to HDLs and Clash is future work.

4.7 Open-source Software

To realize Ex-PART, we either had to implement a logic synthesis and place and route tool from scratch or modify existing tools. Since implementation of such tools is not in the scope of this research we investigated whether it was possible to modify or extend existing tools.

Since the changes that needed to be made to implement Ex-PART's semantics were quite large there was a good chance that some part of the tool might break or start behaving weirdly due to the changes. When this happens in a closed source tool, there is no way to solve the problem short of talking to the manufacturer and hoping they might help you. When the source-code is available however, we are able to fully investigate where this behavior comes from. Therefore we decided to use the open-source logic synthesis tool Yosys and the place and route tool nextpnr [4]. These tools are the only open-source synthesis tools that can synthesize a design for a real-world FPGA. Inspection of the source code and addition of debugging print statements were applied at least twice in the project to debug Ex-PART.

Further facilitating implementation of Ex-PART is the input and output formats of these tools. The output of Yosys is a JSON file which is also the input of nextpnr. By modifying the JSON file Yosys outputs before giving it to nextpnr we can inject our own processing. The implementation of this processing is further aided by the fact that Yosys' outputs are very well documented [5].

Additionally, nextpnr can be extended with Python scripts. Being able to use a modern, fully fledged, widely used programming language to process data in between the steps nextpnr takes enables a great flexibility.

4.8 Compilation Strategies

The design of Ex-PART is highly influenced by realizability. It is a high priority to be able to demonstrate that synthesis like this works for real-world FPGAs. To realize Ex-PART several compilation strategies were considered and tested on their realizability.

To implement the design of Ex-PART for a real-world FPGA the tools Yosys and nextpnr [4] are extended. The organization of these tools has a major influence on the possibilities for the realization of Ex-PART. The most important goal is a tool that can produce designs which are placed as annotated by the designer according to the defined semantics of Ex-PART. Synthesis speed is a secondary goal.

A strategy under consideration is to apply logic synthesis and place and route for every bottom-level component. Logic synthesis has to be applied once for each component. Placement and routing is applied for every *instantiation* of a component. The idea is to run nextpnr for every instantiation for a newly generated FPGA architecture consisting of just the subset of tiles of the ECP5 onto which the component may be placed according to the annotation. Thus for every component it is impossible to be placed and routed outside the allocated area, as nextpnr does not know about any other tiles. To produce the final design these configurations are combined, and between their interfaces routes are set up. This strategy has the upside that it keeps every place and route problem small, which can speed up placement and routing immensely as these problems are NP-hard [1]. This approach is sadly unpractical due to nextpnr's organization. Generating a new FPGA configuration for each component seems to require a full recompilation of nextpnr. This is not easy to manage and takes a long time.

The strategy we did use was extending nextpnr by a Python script right before the placement step. Such a script gets access to a `ctx` (context) variable containing all the nets, cells, routes, placements, and whatever else is relevant. The context object also contains functions allowing safe modification of the state. One of these functions is `createRectangularRegion` which defines a rectangular region on the FPGA identified by a name. Using `constrainCellToRegion` a cell can be constrained to a region by the region's name. Ex-PART creates rectangular regions for every component instantiation, and then every cell is constrained to the correct region. Note that this strategy adds constraints to the placer. Therefore, we cannot make any claims on place and route speed: tight constraints may take longer to place, while loose constraints can speed it up. As the placement has a huge effect on the routability of the design, placement constraints may also have an effect on the time the routing step takes.

4.9 Routing

Ex-PART's low-level directives are limited to placement. Another obvious place were low-level directives can be useful in FPGAs is routing. In this section we elaborate on why Ex-PART does not enable control over routing, and give some of the ideas on routing we did have.

As mentioned in Section 4.1, we first built a hypothetical toolchain where the lowest level describes designs for the ECP5 FPGA on as low a level as possible. On that level, routing is done by configuring routing switches: every slice has some multiplexers that can be configured and thus every signal can be routed via these multiplexers to a different slice on the tile or a different tile altogether. Since these multiplexers are baked into the chip the options here are quite limited and very specific. For example, on the ECP5 it *is* possible to route a signal to a tile six tiles south via one switch, but it is not possible to do so to a tile five tiles south.

As the goal was a *productive* low-level language there had to be some kind of abstraction on top of this routing system as for complicated designs it becomes impossible for a human to keep track of all the multiplexer settings and what that results in. Upon further inspection of the place and route tool it turned out that it did have features to constrain placement, but not for routing. Since constraining placement was considered a good demonstration of a low-level feature we decided that Ex-PART would not provide any low-level programming constructs to constrain routing.

Some features that slightly constrain routing were considered but never implemented. In the first design of Ex-PART it was disallowed to connect any two components if their areas did not share a border. That meant that components that had to be connected had to be placed next to each other by the designer, and the router would have the additional information that the routing would have to go through that border. A problem that arose later that showed that this would not have worked is the fact that the designer does not have any way of specifying where in the rectangle the interface of the component is synthesized. Therefore the designer might put some component B to the right of a component A, while the output interface of component A is on its left side. Often the placement tool is able to infer that it should place the interface of a component in the direction of the component that interfaces with it, but this does not always happen as exactly as one would like (especially with large bit width interfaces).

A design that is legal in Ex-PART but is very hard to synthesize exists: define a mealy machine with a ridiculously high bit width input and output without much logic between them, and instantiate it. This component will utilize a lot of routing resources to route that input to the output, without using many LUTs. The analysis script will show that this component uses hardly any LUTs which may lead the designer to erroneously think that this component uses little space. Here the syntax of the language does not discourage an inefficient choice, however the routing visualization in the visualizer should give the designer some idea that what they are doing is very hard to synthesize. Furthermore, since Ex-PART is explicitly designed as a low-level language it is considered reasonable to assume that designers using Ex-PART are very familiar with their target FPGA and all its available resources.

Chapter 5

Software Manual Introduction

Introduction

*Ex*plicit *P*lace *A*nd *R*oute *T*ool: a low-level hardware description language for FPGAs allowing fine-grained control over placement of modules.

As this software was written for academic research, no time was spent on a user-friendly interface, clear errors, or any other feature one may expect from maturely developed software. Your best hope of working with this project is opening it in GHCi and looking around the code when errors are encountered. Furthermore, much information is dumped in the output directory of the synthesized project. Errors not given by this tool may have ended up in a .err or .log file. This manual aims to give a comprehensive overview of any error you may encounter and of every file involved in the project.

Getting Started

The following chapters explain everything there is to programming in Ex-PART, and to maintaining the software. Ex-PART repurposes Clash' syntax for many of its constructs. Therefore it is good to be at least slightly familiar with Clash. Ex-PART is written completely in Haskell, given that Ex-PART is very much immature software, experience with reading Haskell code, and some understanding of its type system is strongly advised. Although the Haskell features used here are quite elementary, here are some subjects this project uses that you may need to brush up on: Parsec, Records.

In the first chapter on setting up the software you will find the necessary versions of all the software Ex-PART depends on, and information on how to best structure and build an Ex-PART project.

After the set-up guide the guide for programming in Ex-PART is presented. It contains documentation for every language construct, tips on how to best configure your editor, and warnings about everything that seems like it should work but doesn't.

In the examples section several examples detailing all Ex-PART's features are located. The examples in the paper are `collatz`, `md5_reuse`, and `manycore`. Explanations of what all the available examples are supposed to do are located in this section.

Next is the Ex-PART maintenance manual. If you want to add a feature to Ex-PART, read this guide to discover where that can be done. It also contains an explanation of the project structure.

As Ex-PART was developed for a master thesis, there are still many issues with the software. Inspect the GitHub issues page [6] to find what bugs are present and which enhancements are possible.

In case the software does not run, one build directory (`collatz`) is committed to the repository, such that at least one run of building a project is available for inspection.

Notes on the Paper

In the paper *On Productive, Low-Level Languages for Real-World FPGAs* (Chapter 3), a hierarchy of component instantiations and connections is called a *module*. In the code and explanation in this repository it is called a *system*. The name module is used in the paper as it ought to call up the concept of Verilog modules, which are slightly more general than Ex-PART's systems.

Notes on Printed Versions of this Document

This document is an adaptation of the Markdown version available in the repository [7]. This Markdown version contains links to code files, chapters, and GitHub issues which can make everything considerable easier to understand as it is possible to conveniently navigate to relevant documents. When a GitHub issue is mentioned this is done as "#6", which refers to `https://github.com/PietPtr/Ex-PART/issues/6`.

Chapter 6

Setting Up

6.1 Prerequisites

Ex-PART is built on exactly these versions of software involved, any older or newer may work, but probably won't as Ex-PART uses features which are changed often. Find here all used software and their `--version` strings.

Yosys version 0.10.0:

```
Yosys 0.10.0 (git sha1 UNKNOWN, gcc 11.1.0 -march=x86-64 -mtune=generic -O2 -fno-plt
-fPIC -Os)
```

nextpnr at git hash `dd637643`

```
nextpnr-ecp5 -- Next Generation Place and Route (Version dd637643)
```

Ex-PART uses nextpnr's Python script extension, so make sure you enable Python when building nextpnr.

Clash version 1.4.6

```
Clash, version 1.4.6 (using clash-lib, version: 1.4.6)
```

Boost 1.76.0-1

```
extra/boost-libs 1.76.0-1 (2.3 MiB 9.3 MiB) (Installed)
```

Python 3.9.7

```
Python 3.9.7
```

Pygame 2.0.1

```
pygame 2.0.1 (SDL 2.0.16, Python 3.9.7)
```

6.2 Project directory structure

A project called 'project' is best structured as follows:

```
project/
    project.expc
    project.expi
    project.lpf
```

It is possible to use different names for the expc/expi/lpf file, but using the same allows the use of the shorthand `make` function to build a project.

6.3 Running Ex-PART

Make

First run either `make install` or `make symlink` with root privileges. This will create the directory `/usr/share/ex-part`, and either copy or symlink certain files (scripts, CSVs) there. Ex-PART always assumes that this directory exists and that the relevant resource files are located there.

ghci

Ex-PART does not come with a nice `main` function, instead it is generally run through `ghci`. To run it, navigate to the root directory of this repository and run:

```
ghci -iparser -iclash-generator -ijson-builder -iyosys -inextpnr
    -ielaboration -icompiler Main.hs
```

As far as I am aware, Ex-PART does not have any dependencies that do not ship with Haskell by default. A nicer way to structure and run all this is by using Cabal or Stack, see issue #6.

6.4 Flows

Running flows

Ex-PART comes with several 'compilation flows', these are basically build scripts that build the project using different approaches. All flows can be found in `/compiler/Compiler.hs` and have type `Flow`. A flow can be run by itself, it always requires four arguments: the `.expc` file, the `.expi` file, the constraints (`.lpf`) file, and the output directory. If the first three files have the same name (as suggested in the previous section) they can be run with the `make :: Flow -> String -> IO ()` function, which takes a flow, a project name of one of the projects in the examples, and runs the flow for that project.

Compile flows often change directory. When a compile flow crashes the directory is not reset to the main directory. Use `:r` to reset this. This is of course not very convenient, see issue #8.

List of Available Flows

Here all available flows are listed, including their caveats and features.

`clean`

The clean flow rebuilds the entire project. It clears the output directory of all files generated by previous runs and rebuilds everything from scratch.

`auto`

Since the clean flow can take quite some time (rerunning Clash on all components takes forever), the auto flow can be used. The idea of this flow is that it detects which components must be rebuilt (if any), and if the connections have to be rebuilt.

This detection is not always very good, as detailed in #7.

`monolithic`

Builds the project as a monolithic design: it produces Clash.hs (normally used for synthesis), which has every function inlined by default. This file is converted to one enormous Verilog file with clash, synthesized with Yosys, and placed and routed without any placement constraints by nextpnr.

The build files for the monolithic flow are stored in the `monolithic/` directory in the given output directory.

`hierarchic`

Similar process as the monolithic flow, but adds annotations to Clash.hs to *not* inline every function. This produces a hierarchic Verilog project that is synthesized by Yosys and placed and routed without placement constraints by nextpnr.

The build files for the hierarchic flow are stored in the `hierarchic/` directory in the given output directory.

`resource`

For every component in the design, produces a clash file, synthesizes that (so just the component is synthesized), and places and routes it using the `--out-of-context` switch of nextpnr. This option does not route the ports of the component to I/O ports. This flow is used to investigate resource usage of the components used in the design to provide information necessary during layout. This is the flow where you would discover how much area one component needs (at least).

`resource'`

The same as `resource`, but additionally takes a list of strings (so call it as e.g. `make (resource' ["component1", "component2"]) "project"`) of the names of the components for which it should run the resource flow. This is a nice flow to have when you do not want to re-synthesize every component after adding some new ones.

`sim`

Only rebuilds `Clash.hs`. Useful when the expc/expi has changed and only the functional correctness of the design must be checked.

`location`

Only rebuilds `location.json`. This may be useful when the auto flow seems broken (#7) and only the expi file has changed.

`pnr`

Only runs nextpnr on the project. Assumes that `locations.json` and `synthesized.json` are both available. Useful when placement has failed and a subtly different placement is tried (in tandem with the `location` flow).

Chapter 7

Programming In Ex-PART

7.1 Designing Hardware with Ex-PART

In this guide as much as possible information on how to *design hardware* in Ex-PART is provided. Since Ex-PART was developed for a Master thesis, there are quite some hacks and pitfalls you may encounter and there are a wide variety of features available.

This guide fully focuses on allowing you to design hardware in Ex-PART. If anything goes wrong (crashes, unexpected results, etc), take a look at the maintenance manual to find out how to continue, and where you may need to apply a fix.

7.2 General Remarks

- Haskell syntax highlighting works quite well with Ex-PART, it's not perfect but as there was not really a consistent grammar available no custom syntax highlighting was developed.
- Parse errors are usually not very clear, see issue #10.
- Parse errors may occur when spaces or other white space occurs in the wrong place. The parser does not allow some kinds of statements to end with white space, for example.
- Parse errors will also occur when the character '}' occurs unexpectedly, since this character is used to detect the end of a block (like a `haskell` block, as will be explained later). This character occurs in Haskell multiline comments and in records, hence these cannot be used in Ex-PART.
- If parse errors in Haskell statements happen because of other reasons, it's possible that you used a character that is not mentioned in `haskell_stat` in `Parse_shared.hs` (See also issue #1)
- Bit widths of types are not determined automatically. In `parser/Types.hs` a switch statement is located that matches the name of a type to a bit width. Usually this becomes clear by the error "cannot find birdbath for type `<Type>`". This means that if you use any type that it is not listed there, or define a type synonym of the same name but with a different bit width that is defined there, Ex-PART will not function correctly (Issue #2). If you want to add a type, either solve this problem correctly by looking at how Clash determines type bit widths, or simply add / modify the case statement.
- Scopes are very unclear. No time was spend on designing some exact scoping of variables, and in practice there are several namespaces in Ex-PART (even though they were not necessarily built on purpose). There is the namespace of component/system instantiations, these are given a name such that they can be referred to in coordinate expressions. These names are globally referenceable in

coordinate expressions, but no error will be given if they do overlap, simply the 'first' will be returned. Then there is a namespace of system and component type names, which is used to designate the kind of system instantiated when instantiating something. This is also globally referenceable. If anything goes wrong in scopes, it's likely that there is a bug, try renaming variables such that everything has a unique name and see if that goes better. Furthermore, I have not checked whether scopes in the simulation and in synthesis behave the exactly the same. See also #11.

- The feature list below also details some hacks, tips, and tricks for these features that may ease development or help understanding why something behaves unexpectedly.
- When driving the same input with two outputs, none of the tools will give warning, the design is just synthesized to … something. Beware.
- Overlap between components is allowed in Ex-PART. Experience shows that it is often very useful to have two components share area; if they are thoroughly interconnected it is much easier to find a good placement if the LUTs can be in the same rectangle.

7.3 Glossary

- **Component**: a mealy machine defined in the component file (`.expc`).
- **Element**: Anything with input and output ports that can be laid out on the FPGA: so either a component or a system.
- **Instance**: a particular occurrence of some element, with a position and size, on the FPGA.
- **System**: a hierarchy of systems defined in the instantiation file (`.expi`), corresponds loosely to a *module* in Verilog, or a function in Clash operating on `Signal`s.

7.4 Design Feedback

To aid in designing with Ex-PART, some helper tools are available. These give much information on your design *while* you are working on it.

Visualization

The Python program `vizualizer/main.py` is a visualizer for two types of output JSON files of Ex-PART. It can visualize component placement (taken from the generated `locations.json`) and placement of slices onto the FPGA taken from an output JSON generated by nextpnr (consistently named `bitstream.json` throughout all Ex-PART builds). `visualize.py` expects one argument: the directory in which it should search for JSONs. When none are found or the directory does not exist, the program simply shows an empty screen. As soon as either one of the JSONs is available it will be drawn by the visualizer. To differentiate between location information as given in the `.expi` and placement information as generated by Ex-PART, `.expi` colors are more muted and drawn as a border. If the information is available, a legend will be drawn in the top right corner. Indicators in the bottom right corner show which files are loaded.

Example commands

```
python visualizers/main.py project
```
Shows bitstreams and locations in project directory `project/`, as soon as they are generated.
```
python visualizers/main.py project/builds/component/
```
Shows out-of-context (i.e. I/O is not routed to) pnr result of component `component` in project `project`, as soon as it is generated.
```
python visualizers/main.py project/monolithic/
```
Shows bitstream as generated by the monolithic flow for project `project`.

The visualizer can always be run and does not need the directory to exist, it simply waits until the directory contains visualizable files. File loaded indicators in the bottom right corner indicate which visualizable files were found.

Controls

Click-and-drag works to pan around the design. Scrolling zooms in and out. Pressing `F1` re-randomizes all the colors (can be useful if some colors look too much alike or are just plain ugly). `F2` takes a screenshot and saves it as "<unix-timestamp>.png". Pressing `R` sets the coordinate ranges to relative mode: the tile your mouse hovers over is then (0, 0). This feature is currently broken if the view was panned (#23)

Notes:

- The visualizer will try to visualize any directory that contains either a `locations.json` or a `bitstream.json`, so it also works on the output directories of the monolithic, hierarchic, and resource flow.
- There is a switch, -c, to enable showing connections that leave a component. This will render every connection that goes from one cell in a component to a cell in some *other* component.
- There is also a switch, -p, to enable a simpler view that shows the full ECP5 at once.
- The visualizer is built entirely for the 85k version of the ECP5 [8]. For more comments see issue #9.

Metric Analysis

All the tools Ex-PART runs to obtain results emit outputs and errors that may be valuable for debugging and evaluating a design. Since these are spread all over the place, a script gathering these metrics is available in the root directory: `analyze.py`. Simply run it with the name of an Ex-PART output directory as its argument and it will search all available logs for metrics like LUT/FF usage, maximum frequency, and run times. If it can't find a metric it will print it as a dash. If hierarchic and monolithic output directories are available it will print the metrics in a table to easily compare them. Just as the visualizer, this script can be run on directories in the `builds/` directory to gain insights into resources used by components.

The Output Directory

A ton of information is available in the output directory, all the logs, intermediate files, simulation files, etc. If anything goes wrong, do look at those files as some errors may have ended up only there and not in the output of Ex-PART.

`Clash.hs`

For every project a `Clash.hs` file is generated, this is intended for simulation, but running it with Clash can also catch type- and other errors.

7.5 Thorough Explanation of a basic Ex-PART program

In this chapter we will walk through the collatz example (`examples/collatz/`). This file describes a piece of hardware that keeps a number in a 16 bit register, and applies the rules of the Collatz conjecture, i.e. if the number in the register is even it is divided by two, and if it is odd it is multiplied by three and one is added to it.

In the image below the architecture is shown. Every box is a mealy machine as will be defined in the component (`.expc`) file, and every line is a connection between I/O ports of the components. Lines are annotated with the *type* of the output and input port they connect.

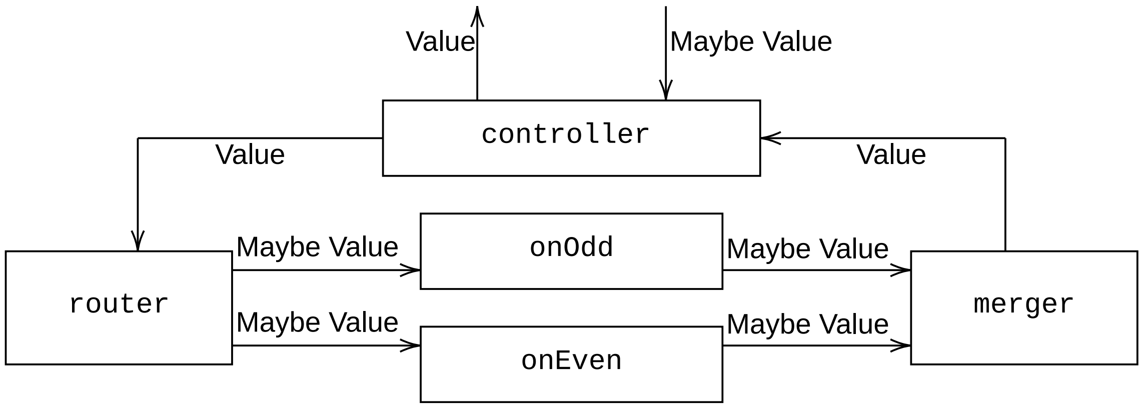

Figure 7.1: Collatz conjecture calculator architecture

`.expc` file

Now, an explanation of the `.expc` file, starting at the top:

```
haskell {
    (>>>) :: Bits a => a -> Int -> a
    (>>>) = shiftR

    (<<<) :: Bits a => a -> Int -> a
    (<<<) = shiftL

    type Value = Unsigned 16
}
```

In a `haskell` block arbitrary Haskell code can be added to the design. Each component has access to these definitions. Do not indent them as they are copied straight to a Haskell file. Define helper functions, type and data definitions, and debugging functions here. By default, `Data.List` and `Clash.Prelude` are available, and the extension `NumericUnderscores` is enabled.

```
component router() {
    input val : Value
    output odd : Maybe Value
    output even : Maybe Value

    even = if testBit val 0 then Nothing else Just val
    odd  = if testBit val 0 then Just val else Nothing
}
```

A component definition. After the keyword `component` the name of the component is defined, followed by `()`. The parentheses are necessary as a feature was planned to allow generics to be passed to a component. This feature has not been implemented, but some support for it remains there in the parser (issue #19).

The first lines of a component are the input, output, and state definitions. This component happens to be a combinational component: it has no state. This component receives a 16-bit value (namely `val`, of type

`Value`, which was defined to be an `Unsigned 16` in the `haskell` block earlier). It has two outputs, both of type `Maybe Value`.

```
component onEven() { ... }
component onOdd() { ... }
component merger() { ... }
```

These components are also all combinational, so not much news happens here, so their implementation is omitted.

```
component control() {
    input next_val : Value
    input set_val : Maybe Value
    state last_val = 0 : Value
    output result_value : Value

    last_val' = case set_val of
        Just new_value -> new_value
        Nothing -> next_val

    result_value = last_val
}
```

This is a component with state. Its state is defined in the fourth line. It is given a type just like the inputs and outputs (`Value`). Additionally an initial value is supplied, namely `0`.

To define the state transition, an expression is defined for `last_val'`. Notice that this expression can depend on any of the inputs, and the previous state. It can also depend on some other state, or their next states, as long as they do not form a mutually recursive dependence.

`.expi` file

With the components defined, the `.expi` file can be written to layout those component on the FPGA.

```
system in (6, 5) at (2, 2) {
```

The top-level system is called `system`, takes up an area of six by five, and is located at position (2, 2) on the ECP5. This coordinate system is zero-indexed, (0, 0) is the top left corner. Ex-PART uses the same system as the HTML documentation of the ECP5.

The size of systems is not checked by Ex-PART, if you specify (1, 1) here it may work as well. Where it is taken into account is when any of the components *refer* to this value, that is if a component is placed at e.g. `(system.x, 0)`. This size is currently not in any way inferable, if you resize components in a hierarchy and you need to use an accurate size for the system, you need to update the system size manually. See also issue #12 on inferable sizes.

```
    input setting : Maybe Value
    output result : Value
```

The I/O of the top-level system. As the input we define `setting`, this `Maybe Value` can set the value in the register to which the Collatz conjecture rules must be applied. `control` is a component that has an input of type `Maybe Value` for exactly this purpose, so once that component is instantiated we must route this input to that component.

Since this is I/O of the *top*-level system, this is also the I/O that must be constrained in the `.lpf` file.

```
    controller is control in (6, 1) at (0, 0)
```

The `control` component is instantiated. It is given the name 'controller', an area of six by one, and the location *(0, 0)*. This location is relative to the system it is a child of, so on the FPGA this component will be located at (2, 2).

The size could also have been defined as `(system.w, 1)`, for example.

```
    controller.set_val<-setting
    controller.result_value->result
```

The controller's inputs are linked to the inputs of its parent system. Notice that the arrow notation can go both ways. A port of the control component is referred to by writing the *name* of the instance of the component, followed by a period, and then the port name.

Local system ports do not need this period-syntax, they are simply referred to by name, as is done with `setting` and `result`.

With these statements the system I/O port setting and result are connected to the controller, so that the controller component drives/is driven by the FPGA I/O ports as intended.

```
    collatzer in (controller.w, 4) at (0, controller.h) {
```

A subsystem with the name `collatzer` is defined. At this point it is probably a good idea to view the file this comes from, as the indentation here will make it much clearer that this is a *sub*-system. Note that its size and position is defined in terms of the controller, so if we resize the controller, this entire system moves with it.

```
        input val_in : Value
        output val_out : Value
```

This subsystem also has its own input and output ports. Since the goal of this system is to calculate the next value of its input according to the rules of the Collatz conjecture, the types of the single input and output port are both `Value`.

```
        router is router in (1, onOdd.h + onEven.h) at (0, 0)
        onOdd is onOdd in (collatzer.w - 2, 2) at (collatzer.x + 1, 0)
        onEven is onEven in (onOdd.w, onOdd.h) at (onOdd.x, onOdd.h)
        merger is merger in (1, onOdd.h + onEven.h) at (onOdd.x + onOdd.w, 0)
```

All the necessary components are instantiated. To showcase what kind of expressions are possible in the coordinate and size expressions they have been written to be highly dependent on other values. Notice that both components (like `onOdd`), a system (`collatzer`), and constants are used in these expressions. See also issue #13 and #14 for more information on what you cannot do with these expressions.

```
        router.val<-val_in
        router.odd->onOdd.val
        router.even->onEven.val
        onOdd.res->merger.vo
        onEven.res->merger.ve
        merger.res->val_out
    }
```

To finish up the subsystem `collatzer` the connections are defined as in the diagram shown at the start of this chapter. If you try to connect ports of different types, Ex-PART will throw an error.

```
    collatzer.val_in<-controller.result_value
    collatzer.val_out->controller.next_val
}
```

The I/O ports on the subsystem need to be connected to the controller, and that is exactly what we do here. The ports of the subsystem can be referenced by using the name of the subsystem, a period, and then the port name.

7.6 Simulation with Clash

The following flows generate simulation files: `clean`, `auto`, `monolithic`, `hierarchic`, and of course `sim` (which is there *just* for simulation file generation).

"Simulation files" in Ex-PART are simply Clash files, i.e. Haskell code. After using either one of these flows you will find the files `Definitions.hs` and `Clash.hs`. `Definitions.hs` contains everything you put inside `haskell` blocks, and `Clash.hs` contains all the mealy machines representing components, and functions representing the hierarchies of systems in the `.expi` file. If you're familiar with Clash it may be very useful to inspect these generated files when checking for functional correctness. They have been generated in a way that they ought to be relatively readable. Furthermore, in the output directory a directory called `builds` is created, which contains a directory for every component in the design. These directories also contain Clash files for just those components. These are called `Synth_<component>.hs`, where `<component>` is the component name.

To simulate your entire project, use the command:

```
clashi Definitions.hs Clash.hs
```

To simulate just one component with the name `component`:

```
clashi Definitions.hs builds/component/Synth_component.hs
```

Once in the `clashi` shell, test for functional correctness as usual using Clash's `simulate`, `sample`, etc. functions.

7.7 Post-Synthesis Simulation

Since Ex-PART operates directly on the JSON file that Yosys outputs, no post-synthesis simulation is available.

7.8 Bitstream Generation

Ex-PART uses nextpnr to generate bitstreams. Nextpnr emits bitstreams in both JSON format and the Trellis textual configuration format. Both of these versions are available in the output directory as `bitstream.json` and `bitstream.config`. With the program `ecppack` (which is part of project Trellis) `bitstream.config` can be converted from a textual representation to a bitstream that can be programmed to an ECP5 FPGA.

Nextpnr warns that it is experimental software and that it *might* break your FPGA. Ex-PART is (clearly) even more experimental, and uses nextpnr. Before flashing any designs Ex-PART generated to your FPGA, you should really try to manually verify nothing weird is going on in both the `bitstream.json` and the `synthesized.json`.

7.9 Feature List

In this chapter you find a comprehensive feature list of the language Ex-PART. For every feature a brief explanation is provided, syntax examples are given, and the example designs which use the feature are listed. For a short description of their implementation the same list is available in the maintenance manual.

Comments

Comments are a little iffy in Ex-PART. In .expi files single line comments usually work:

```
a is a in (5, 3) at (0, 0) -- this is a single line comment
```

In expc files comments at least do always work in the expression statement parts, as those are directly copied to Haskell code during compilation. However, comments outside components and between input, output and state definitions *may* not work. Multiline comments do not work, as the } character is used to determine if a component definition has finished (issue #31).

```
    -- This comment fails
    component router() {
        input val : Value
        -- This comment fails
        output odd : Maybe Value
        output even : Maybe Value

        -- A working comment
        even = if testBit val 0 then Nothing else Just val -- This comment is fine
        odd  = if testBit val 0 then Just val else Nothing
        {- This multiline comment
           causes problems
        -}
    }
```

Types

In both component definitions and system definitions ports and states need to be annotated with a *type*. These types are exactly Clash types, so it is always possible to give ports any type Clash supports. Do note that the parser of the types in Ex-PART does not support everything, so to circumvent any parse errors you may get in types, define a type synonym in a `haskell` block. For example, this line may not parse:

```
input inp : Maybe (Vec 4 (Unsigned 16))
```

Solve this by defining this `haskell` block:

```
haskell {
type SomeInputType = Maybe (Vec 4 (Unsigned 16))
}
```

And changing the erroneous line to:

```
input inp : SomeInputType
```

`.expc` file

The component or `.expc` file is where components and generally available Haskell code is defined. Every example contains an `.expc` file as without components no hardware can be described.

`haskell` block

In a `haskell` block general Haskell code can be written. Any component can use the function or type definitions defined in such a Haskell block.

In this example two synonyms for the Clash functions `shiftR` and `shiftL` are defined, and a type synonym for `Unsigned 16` is set.

```
haskell {
(>>>) :: Bits a => a -> Int -> a
(>>>) = shiftR

(<<<) :: Bits a => a -> Int -> a
(<<<) = shiftL

type Value = Unsigned 16
}
```

This is useful because now it is possible to easily change the bit width of every port and state in the components simply by changing the type to which `Value` aliases.

Do not indent any code in this block, as Clash will give errors during Verilog compilation or simulation.

Every example uses a `haskell` block since basically always you'll need to define some helper functions and types / type aliases. There may be a bug as well regarding `.expc` files without a `haskell` block: issue #4.

Component definition

A general component definition looks as follows:

```
    component <name>() {
    input <input n>                          : <input_type n>
    state <state n> = <initial_state n> : <state_type n>
    output <output n>                        : <output_type n>

    <state n>' = <state_expr n>
    <output n> = <output_expr n>
}
```

Where `<text n>` means that there can be any number `n` of such statements.

- `<input n>`: the name of some input port.
- `<input_type n>`: the type of some input port.
- `<state n>`: the name of a state.
- `<initial_state n>`: the value for the initial state of that state. The parser is quite weak for this initial state (e.g. `Just 0` may not work), define a constant function in a `haskell` block to circumvent this (e.g. `my_initial_state = Just 0`, and setting the initial state to `my_initial_state`). When you forget this value, a very unclear parse error pops up. When these statements don't parse it's usually because the initial state was left out.
- `<output n>`: the name of some output port.
- `<output_type n>`: the type of some output port.
- `<state_expr n>`: the transition expression for this state. This expression is simply a Haskell expression. Any Haskell construct can be used here, except for records, as they contain the character } in their syntax (see also issue #15). A transition expression can use any of the variables that are in scope in the component: inputs, other states, even the next values for other states, as long as there is no mutually recursive dependency between them. To find out if this has happened (on accident), run the Clash simulation. If it produces no output there probably is such a mutually recursive pair in a component somewhere.
- `<output_expr n>`: the transition expression for some output.

Notice that there is a prime (`'`) after `<state n>` in the equation for the transition expression! The new state for an old state `state` is always called `state'`.

Example components are given in the "Thorough Explanation of a Basic Ex-PART Program" section, and of course every example project contains many components.

`.expi` file

In the instantiation or `.expi` file the components defined in the `.expc` file are laid out on the two-dimensional grid of the FPGA.

Coordinates and Sizes

Every system definition and component instantiation has a size and location. These are both given as two-tuples, `(width, height)` for the size and the `(x, y)` for the location. In these tuples *layout expressions* can be used. The operators `+` and `-` are available to define layouts. Furthermore, layout properties of other components can be used. The layout properties for a component called `component` are as follows:

- `component.w`: the width of `component`
- `component.h`: the height of `component`
- `component.x`: the x coordinate of `component`
- `component.y`: the y coordinate of `component`

There exists somewhat of a global scope for these variables: 'somewhat' because Ex-PART does not give errors if any names in this scope overlap, it simply returns the first value it finds.

Parentheses can be used to impose precedence on sub-expressions as usual in arithmetic expressions.

In almost every example many examples can be found of these expressions being used.

See also issue #13 on cycle checking in coordinate and size expressions.

System Definitions

A system definition is of the following form:

```
<system_name> in (<system_width>, <system_height>) at (<system_x>, <system_y>) {
    input <input n> : <input_type n>
    output <output n> : <output_type n>

    <instantiations>
    <connections>
    <subsystems>
    <elaborated_features>
}
```

- `<system_name>`: the name of this system, can be any identifier.
- `<system_width>`: the width of the system.
- `<system_height>`: the height of the system.
- `<system_x>`: the x coordinate of the system.
- `<system_y>`: the y coordinate of the system.
- `<input n>`: some input of the system.
- `<input_type n>`: the type of some input of the system.
- `<output n>`: some output of the system.
- `<output_type n>`: the type of some output of the system.
- `<instantiations>`: component or system instantiations. The order of statements of this and the following three kinds does not matter.
- `<connections>`: connections between ports of systems and components.
- `<subsystems>`: subsystem definitions. A definition for a subsystem is exactly the same as a system, just indented by one more level (by convention).
- `<elaborated_features>`: Some extra syntactic sugar is available to make development of repetitive designs easier, these features are all "elaborated" to a collection of the previous three kinds (instantiations, connections, subsystems).

Top-level systems have some more requirements/caveats: the width, height, x, and y expressions *must* be constant expressions: they cannot depend on any other component or system. Furthermore, the ports defined as I/O ports in the top-level system are the ports that must be constrained to the FPGAs I/O pins in the `.lpf` file.

Component Instantiation

The basic way to layout components is as follows:

```
<instance_name> is <component_name> in (<width>, <height>) at (<x>, <y>)
```

- `<instance_name>`: the name given to this instantiation. May be the same name as the component that is instantiated.
- `<component_name>`: the name of the component that is to be instantiated, this is the name of one of the components in the component file.
- `<width>`, `<height>`, `<x>`, `<y>`: layout expressions defining the position and size of this instance.

Port connection

Ports of instances and systems can be connected as follows:

```
    <from_element>.<from_port>-><to_element>.<to_port>
    <to_element>.<to_port><-<from_element>.<from_port>
    -- Example:
    component.output_port->component2.input_port
```

Notice the `.` and `->`/`<-`. The period signifies that the port to its right is a port of the element to its left. The arrows denote connection. Both directions are available, use whichever is more convenient to read or write at the moment.

An "element" refers here to anything that has ports: systems, components, and subsystems.

Ports of the current system are referred to without any element:

```
<system_input>-><to_element>.<to_port>
<from_element>.<from_port>-><system_output>
-- Example:
system.output->local_output
```

View the example section and code examples for many examples on connection. There is some extended syntax for so-called multi-connections and constant drivers, explained in their respective sections.

Constant Drivers

When some input of a component must be driven by a constant value, you would have to define a mealy machine with one output whose transition expression is a constant. This is quite cumbersome, so shorthand is available to do this:

```
router.x<-(4)
```

Taken from the manycore example, where a router must know its own position, and that value is simply constantly driven to the `x` input port of the router. A constant driver is of the form `(<constant>)`, and may only appear on the dash-side of the arrow (obviously, as otherwise you would route the output of some component to a constant value, that doesn't make any sense).

Constant driver support is quite limited at the moment: it is only possible to drive numbers constantly. It would be preferable to have that be any constant value available in Haskell, but as Haskell allows defining your own data types that would result in a bit more complicated parsing.

If you're willing to give up simulation it *is* possible though. Let's suppose we have a component with a port of type `(Maybe (Unsigned 4))`. Suppose we want to drive `Nothing` on it constantly. Since `Nothing :: Maybe (Unsigned 4)` is represented as '`0....`' (cf. `pack (Nothing :: Maybe (Unsigned 4))` in Clash), by driving the constant `(0)` we will get the intended effect in hardware. This is not simulatable since the Clash code will throw a type-error, as `0` is not of type `Maybe a`.

Another approach that does preserve simulation is defining a mealy machine as follows in the `.expc` file:

```
component constantNothing() {
    output c : Maybe (Unsigned 4)

    c = Nothing
}
```

Instantiate the component anywhere, and connect its `constantNothing.c` port to the port that must be driven by Nothing.

See issue #16.

Examples using constant drivers: manycore, chain, core, constants.

Repeat statement

The repeat statement simply *repeats* a component or system several times. In Clash this corresponds loosely with a map. Its syntax is as follows:

```
repeat <repeat_name> at (<x>, <y>) {
    component = <element> in (<width>, <height>),
    amount = <amount>,
    layout = <layout>
}
```

- `<repeat_name>`: The name of this repetition. This name is referred to in multi-connections.
- `<x>`, `<y>`: The location of the *first* instance in this repetition.
- `<width>`, `<height>`: The width and height of *every* instance in this repetition.
- `<element>`: The component or subsystem name to be repeated. A subsystem can be repeated too, if you have instantiated some system somewhere it is possible to *also* lay it out using repeat as well. For more info see the section on unplaced systems.
- `<amount>`: How often the element must be repeated.
- `<layout>`: either `horizontal`, `vertical`, or `identical`. These describe how the layout must be continued: in a horizontal or vertical line, or placing every component at the same position.

This image shows an example of a vertical layout of some component of size (4, 2), at location (2, 2), with amount five.

The order of the settings does not matter.

Once instantiated, individual components of the repetition can still be addressed:

```
<repeat_name>[<index>].<port>->some_port
```

Where `<repeat_name>` is the name given to the repetition, and `<index>` is a **1-indexed** accessor for the components, **so the first component in the repeat with name `repetition` is `repetition[1]`**. This is 1-indexed since this is *not* an array in memory, this is a line of components laid out in 2D space. When, in real life, some line of objects is laid out, it is most natural to refer to the leftmost or topmost component as the "first", not the "zeroth". Furthermore, with 1-indexed component references the last component index is equal to the amount, which is again quite natural: if there are in total five objects somewhere, you refer to the last as the "fifth" in natural language and hence as `repetition[5]` in Ex-PART.

Accessing more than one component at the same time as possible with multi-connections

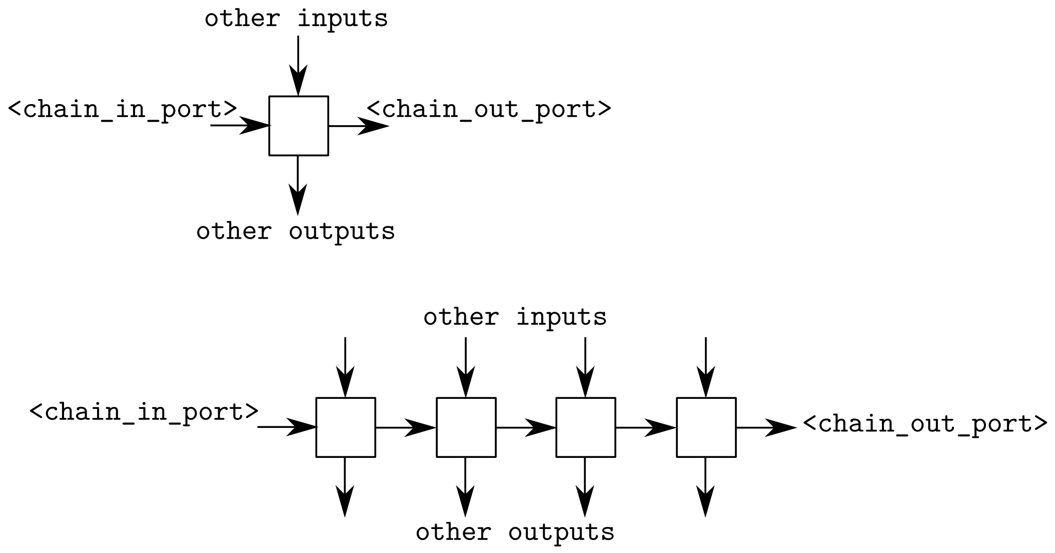

Figure 7.2: Chain diagram

Coordinate and size expressions do not support indexing in the parser. It is possible to circumvent this: internally a statement such as `repetition[1]` is translated to an instance with the name `repetition_1`, and this identifier *is* available in coordinate and size expressions. This hack is used in the manycore. See also issue #17.

Examples using the repeat statement: repeat, manycore, chain, core, smallnet, router.

Chain statement

Use the chain component to build *chains* of components. In Clash this corresponds loosely to a fold. Its syntax is as follows:

```
chain <chain_name> at (<x>, <y>) {
    component = <element> in (<width>, <height>),
    amount = <amount>,
    layout = <layout>,
    chain_in = <chain_in_port>,
    chain_out = <chain_out_port>
}
```

The diagram below shows what hardware this generates.

Given the top component, with an some input port `<chain_in_port>` and an output port `<chain_out_port>` *of the same type*, the `chain` primitive builds a chain of `<amount>` components, connecting the `<chain_out_port>` of component n to the `<chain_in_port>` of component $n+1$.

Other inputs and outputs may be available, as shown in the diagram. These still can be accessed using the same access syntax as in the repeat statement. Accessing more than one component at the same time is possible with multi-connections

Chain is a special (more elaborate) case of repeat, so whatever holds for repeat is usually also true for chain.

This is used in the examples: manycore, chain.

Multiconnections

To easily connect many ports of chains or repetitions of components to other chains and repetitions (to e.g. create a two-dimensional grid of hardware) multi-connections are available. They can take several forms:

```
<from_repetition>:<from_port> -> <to_repetition>:<to_port>
-- Example:
drivers:out -> sum_chain:next
```

Using the `:` with a repetition instead of a `.` denotes that for *every* component in the `<from_repetition>`, the port `<from_port>` must be connected to the `<to_port>` of every component in the `<to_repetition>`. This only works when both repetitions are of exactly the same size. When they are not the following syntax is available to select parts of a range:

```
<from_repetition>[<from_index>-<to_index>]:<from_port>
-- Example:
repetition[1-3]:port
```

This denotes that the `<from_port>` in every component with index larger than or equal to `<from_index>` and smaller than or equal to `to_index` are connected. More examples of this usage are available in the repeat and manycore example.

This is used in the examples: manycore, chain, repeat.

Unplaced Systems

As mentioned, it is possible to chain or repeat a *system* instead of just components. However, to define a system hierarchy implies immediately instantiating it as well. When you just want a chain of the same system hierarchy, and not one extra system somewhere, it is possible to add the qualifier `unplaced` before a system definition:

```
unplaced <system_name> in (<width>, <height>) { ... }
-- Example taken from manycore.
unplaced pru in (28, 14) { ... }
```

This qualifier tells Ex-PART that this system hierarchy may be reused or instantiated *somewhere*, just not here.

This is done in the manycore example.

System Instantiation

Systems can also be re-instantiated using similar syntax to components. This allows you to define a system, and then add several more instances of the system. This is done in the md5_reuse example: first the MD5 hashing hierarchy is defined as the subsystem named `hash1`. Then under that system three statements are located that instantiate three more of the same system to build a design that can perform the same calculation four times in parallel.

```
<instance_name> is <system_name> in (<width>, <height>) at (<x>, <y>)
```

Notice that this is exactly the same syntax as component instantiation. The only difference in semantics is that now we use a *system name* instead of a component name. This `<system_name>` is one of the systems in the `.expi` file, and may be `unplaced`.

7.10 Error Messages

Ex-PART's error system is *very* simple: it uses Haskell's `error :: String -> a` function whenever anything unexpected happens (see issue #20). Occasionally Ex-PART will dump extra output in the error message as well. This list aims to go through all error messages Ex-PART may throw and provide a short explanation on why this error may be thrown and what can be done to fix it.

If an error is not mentioned here, it is probably in the maintenance guide, as it may be indicative of an error in Ex-PART instead of in your `.expc` or `.expi`.

Errors List

- `No such file or directory: /usr/share/ex-part/<filename>`
  - Make sure you have run `make install` or `make symlink`, as explained in setting up.
- `clash-generator/Flattener.hs:131:` No connection specified for element $name (is $type), port $portname
  - The flattener searched for a driver for the port $portname of element $name (which is of component or system type $type), but could not find it. Check if that port is indeed connected to something.
- `clash-generator/Flattener.hs:151:` No connection specified for io statement $io in system $sysid
  - Flattener couldn't find a driver for the IO port $io of a system called $sysid. Also prints all the connections that it did find. Check if that port is connected.
- `compiler/Compiler.hs:91:` No components in expc file...
  - An .expc file must contain at least one component.
- `elaboration/ElaborateConnection.hs:9:` Cannot connect ports $from -¿ $to as they have differing types: $fromType -¿ $toType.
  - Ports must have the same type when they are connected, this error is thrown when two ports are connected with different types.
- `elaboration/ElaborateConnection.hs:25:` Cannot find port with name $portName in $iostats
  - The port with name $portName was not found in the IO statements of an element. It prints the IO statements it did find.
- `elaboration/ElaborateConnection.hs:27:` Found several ports with name $portName in $iostats
  - Only place in Ex-PART that actually errors when several entities of the same names are found, instead of just picking the first one. There are several ports with the name $portName and there should not be. To help debugging, the IO statements that were searched were found.
- `elaboration/ElaborateConnection.hs:40:` Cannot find element with name $name in $elemnames
  - During Yosys post-processing a bit width of ports must be found, and that can be quite hidden in the data structures. That's why these elements must be searched through and these kind of errors may be thrown. If an element that does not exist occurs in e.g. a connection statement this error may be thrown.
- `elaboration/ElaborateConnection.hs:42:` Found several components with name
  - Similar issues but with several components with the name.
- `elaboration/Elaboration.hs:76:` Cannot find system $name in this scope. $sytem_names
  - Is thrown for system instantiations that refer to systems that are not in scope. Check if the system is in scope or you've given your instantiation/repetition statement the correct type. Also prints a list of names of systems it did find.

- `elaboration/Multiconnection.hs:8:` Cannot connect differing amount of ports: $from' -¿ $to'
  - Multi-connections can only connect ranges of the same size. Take care that the ranges you tried to connect are indeed of the same size. This can be especially obfuscated when not using the range operator.
- `elaboration/Multiconnection.hs:20:` Cannot find repetition with name $repname for multi-connection $repname:$portname
  - The specified multi-connection refers to a repetition with $repname, the system did not find any repetition with that name in scope. Check if you are referring to the correct repetition.
- `elaboration/Repetition.hs:34:` Missing option `chain_in` in chain statement
  - Chain statements *must* contain a `chain_in` option (Chains)
- `elaboration/Repetition.hs:37:` Missing option `chain_out` in chain statement
  - Chain statements *must* contain a `chain_out` option (Chains)
- `elaboration/Repetition.hs:42:` Missing option `component` in repetition statement.
  - Repeat and chain statements *must* contain a `component` option (Chains, Repeat)
- `elaboration/Repetition.hs:45:` Missing option `amount` in a repetition statement.
  - Repeat and chain statements *must* contain a `amount` option (Chains, Repeat)
- `elaboration/Repetition.hs:48:` Missing option `layout` in a repetition statement.
  - Repeat and chain statements *must* contain a `layout` option (Chains, Repeat)
- `elaboration/Repetition.hs:119:` Cannot find element $elemName in source files.
  - The system searched for an element of a certain name but couldn't find either a component or a subsystem of that name. Check if you spelled the system or component name correctly in every repetition.
- `elaboration/Repetition.hs:172:` Unknown layout procedure.
  - The only available layout procedures are `horizontal`, `vertical`, and `identical` (Repeat). Use only those, or implement a new one at this line.
- `json-builder/Locations.hs:154:` Could not find ID $id in provided list.
  - The identifier $id in a coordinate or size expression could not be found. Make sure that the identifier was typed correctly and is indeed defined in the instantiation file.
- `json-builder/JSONBuilder.hs:21:` Top-level coordinates must be constants.
  - As defined in System Definitions, the top-level system must have constant coordinates. Ex-PART found a non-constant coordinate.
- `json-builder/JSONBuilder.hs:32:` expi file contains a cyclic coordinate definition, cannot generate location JSON.
  - Ex-PART found a cyclic dependency for coordinates or sizes, see issue #13 for more information on seemingly resolvable dependencies.
- `nextpnr/Nextpnr.hs:28:` nextpnr terminated with code $code
  - Somewhere in nextpnr an error occurred, usually this is a failed assertion, a segfault, or some error in the python script (`nextpnr/constrainer.py`). In any case, take a look at `nextpnr.err` for more information.

- `parser/Types.hs:149:` Cannot find bit width of type $type
  - Bit width for types are hard-coded in Ex-PART, as that was the fastest solution for now. It *should* use Clash's system that maps types to bit widths (See also issue #2). If you want to add a type's bit width, add it to the case statement here.
- `yosys/Postprocessing.hs:204:` Could not find driver $cid in $(sys_connections system)
  - $cid is some connection ID, so an element and a port. In the connections in the current system, no driver driving this connection ID was found. It also provides a list of connections of the system so you can see which connections *were* found.
- `yosys/Postprocessing.hs:331:` cannot find net for cid in netmap $cid ($netmap)
  - Errors here are harder to debug and more often they are errors in Ex-PART and not in your code.
- `yosys/Postprocessing.hs:333:` No net found, something is disconnected... $port ($relevantConnections) ($netmap)
  - Some output port $port is not connected to anything. Much debug output is printed here as well, so you may miss the error because of all the extra output.
- `yosys/Preprocessing.hs:63:` Found zero-output component.
  - Components must have at least one output.
- `yosys/Yosys.hs:54:` Clash terminated with code $code
  - Clash terminated with an error. Take a look at the `clash.err` in one of the directories in `builds/` in the output directory.
- `yosys/Yosys.hs:138:` Yosys terminated with code $code
  - Yosys had some error, take a look at the `yosys.err` and `yosys.log` in the output directory.

Chapter 8

Example Programs

8.1 Examples

In the examples directory you will find several examples developed to test Ex-PART features and to research how well the ideas Ex-PART tries to implement work. Here we detail what their use is and what features they employ. The chapter titles here correspond with (and link to) the directories the example is located in.

For every example a demonstration of how to run it in `clashi` is provided.

To best understand this document it is recommended to first read the programming manual.

8.2 Paper Examples

These three examples are used in the paper.

`collatz`

The Collatz conjecture calculator. In both the programming manual and the paper this example is explained in much detail.

```
*Main> simulate @System system
    [Nothing, Just 5, Nothing, Nothing, Nothing, Nothing, Nothing]
[0,0,5,16,8,4,2,1,*** Exception: X: finite list
```

`md5_reuse`

Most readable code of the four-way parallel MD5 hash calculator. `md5_parallel` describes exactly the same hardware, had the system copied manually to instantiate four hashers. In `md5_reuse` system instantiation is used instead.

This design implements four parallel components that compute the MD5 hash of incoming 128-bit messages. Every cycle 32 bits of a message can come in and are stored until the entire message has arrived. When an entire message has arrived, the message is transferred to one of the hashers in a round-robin fashion by the `load_balance_4` component. 64 cycles after the entire message has been transferred, the hasher will emit the 128-bit hash in four 32-bit messages and the `cat_4_maybes` makes sure exactly one hash is put on the output. Since every hasher takes the same amount of time, the order of the hashes on the output is exactly the same as the order on the input.

The pseudocode shown on Wikipedia was followed. Since the messages are always 128-bit, the outer for-loop is unnecessary: everything is already in a 512-bit chunk. The inner for-loop contains 64 steps that compute the hash by (inter)changing the 32-bit variables A, B, C, and D. To achieve this in hardware, these

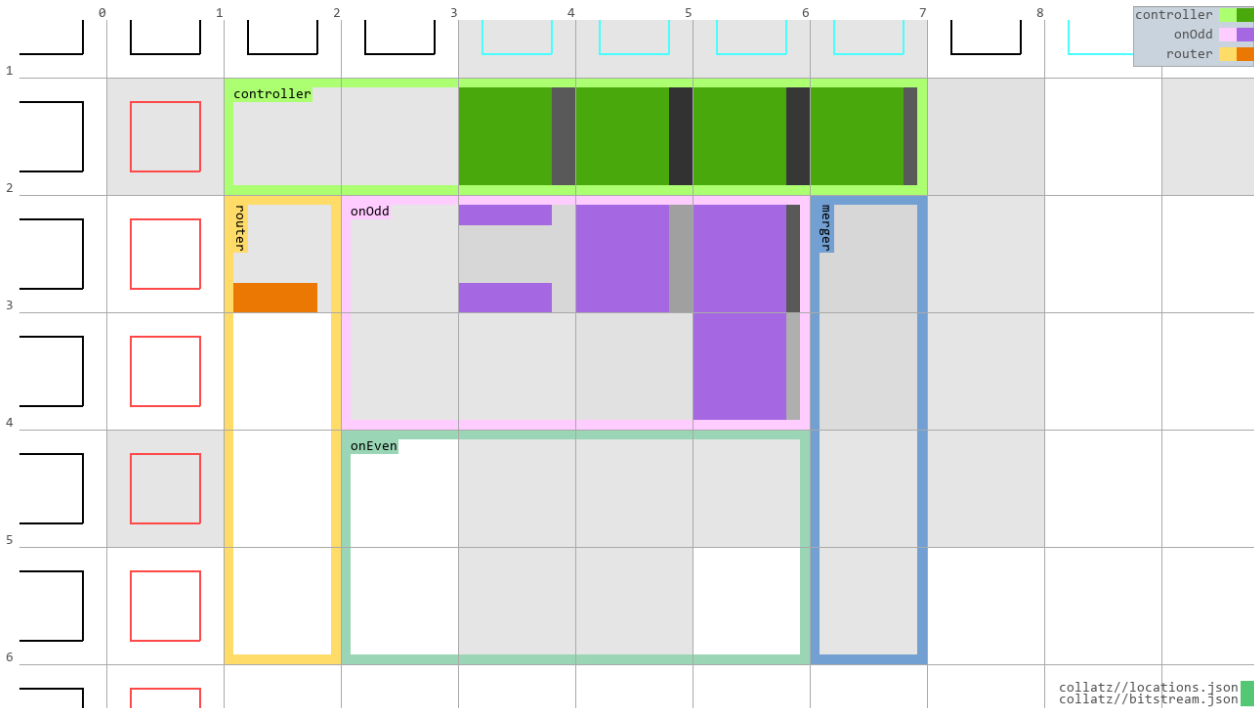

Figure 8.1: collatz as seen in the visualizer

variables are stored in four registers. For 64 cycles, their values are sent into a multiplexer, and based on a cycle counter (`i` in the pseudocode) their values are sent to the correct computation path. To perform these computations, constants are necessary. Two constant stores, for `K` and `s`, are included in the system. The correct value is selected from the store based on the cycle counter (this is also why every instantiation of the MD5 hasher has its own store, it needs a value every cycle, and we would need 4-port read memory to achieve this with just one memory). At the end of the process the found values for A, B, C, and D are added to constants defined by MD5 (a0, b0, c0, and d0 in the pseudocode) to produce the final result.

```
*Main> simulate @System system $
    [Just 0, Just 0, Just 0, Just 0, -- First message
    Nothing, Nothing, -- pause
    Just 1, Just 0, Nothing, Just 0, Just 0] -- second message with break in between
    L.++ (L.take 69 $ L.repeat Nothing)
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Just 869400926,Just 2891193117,Just 428080646,Just 1142714583 -- First hash appears
,Nothing,Nothing,Nothing
,Just 540878663,Just 3516292304,Just 2802766885,Just 4124385854 -- Second hash appears three cycles later
,*** Exception: X: finite list
```

The message consisting of four words of zeroes is loaded in (128 bits, all zero), and 64 cycles later the

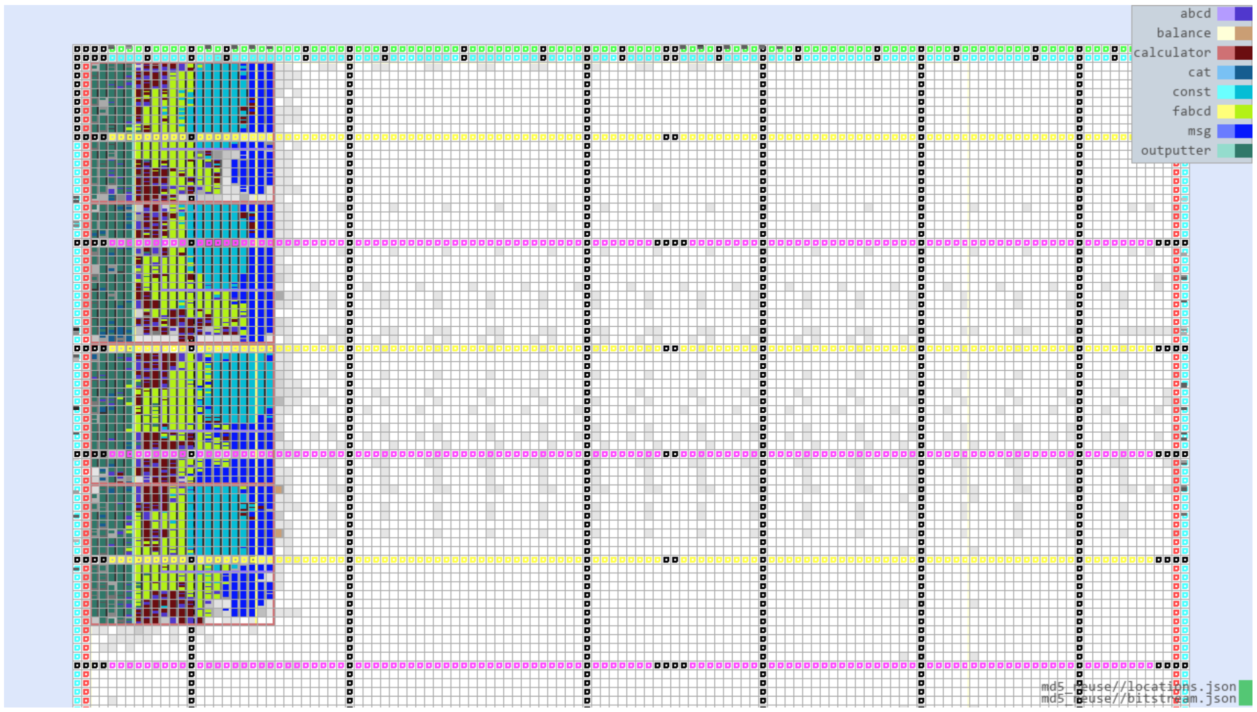

Figure 8.2: md5_reuse as seen in the visualizer

hash appears. Two cycles after the first message, a different message (with just one bit set) is loaded in, with a one cycle pause between the second and third word. This second message is sent to the second hasher and both hashes are computed in parallel.

`manycore`

The manycore as described in the paper. The basic idea of the manycore is that it is a grid of "PRUs": Processing and Routing Units. These units contain a very simple processor and a router. Routers receive packets from their core, and send them to the correct router based on a 4-bit x and y coordinate. When a router receives a packet destined for that location, it sends it to its core.

The `haskell` block of this design is quite large. It contains data types both for the instructions for the processor, and for instructions to the FIFO. Many type synonyms are defined here as well. Two functions, `decode` and `encode`, convert 8-bit representations of instructions to and from the `Instruction` data type. The `Instruction` data type is 10 bits, but we only need 8 to actually store all the information we need. A lot of test inputs for several components is present in this `haskell` block too, and the program (`default_prog`) that is burned into the core (cores are not reprogrammable, the program memory is synthesized into the design). Initial states for many states used in components are defined as well as functions with no arguments and with names like `empty_*`. To do easy conversion from locations on the grid (two 4-bit numbers in a tuple) to 8-bit numbers (the type that goes over the bus between routers), the functions `int2loc` and `loc2int` are used. Data types for states of components are defined here as well.

Components defining the processor are:

- `datapath`: Contains the ALU and logic for manipulating incoming data from the registers, and calculates new values for registers. Also manages program counter updates (both the regular increments and jumps), and sends commands to the FIFO managing outgoing packets of this core.

- `registers`: The register file of the core: four 8-bit registers are available. Register 0 is constant zero when read from, and discards values when written to. Allows selection of two registers in one cycle.

- `prog_mem`: The program memory, simply always sends the value pointed to on its input to its output. As its state is never modified, it can be synthesized as a combinational circuit. The size of this circuit depends on the program burned in the core. The default program waits for inputs it receives (`ReadFIFO` is blocking), and then adds it to a summing register. When it receives the value zero it sends the result of the sum to (4, 5), i.e. the output of the manycore.

Components for managing the outgoing packet queue and the incoming data queue between the router and the processor are:

- `in_fifo`: The FIFO for incoming data from the router. Code taken from an an example given by Clash [9].

- `queue_controller`: `in_fifo` uses quite a different style of arguments from the rest of the system, so this FIFO is controlled by this component. It translates `FIFOCommands` from the processor (to read data) and the router (to push data) to correct signals for the `in_fifo`.

- `packet_queue`: Very similar FIFO to in_fifo, just with some types changed (sadly means that some code had to be copied, as Ex-PART does not allow for most of the pretty abstractions that Clash does).

- `packet_control`: similar job to `queue_controller`, just with the packet queue. A large difference is that the packet queue is a queue of type `Packet`, which is built up by two writes from the processor: first for the 8-bit coordinate (4 bits for x and 4 bits for y), then for the 8-bit value. This component saves the location and waits until the processor has pushed the value, then it pushes the entire assembled packet to the `packet_queue`.

Routers send packets in two cycles: one for the address and one for the data. Every router starts with the same state, and is thus synchronized. Therefore every router knows exactly what to expect on what cycle: either two cycles of no data, or one cycle of location data and one cycle of value data. The router is built from the following components: - `pkt_ser`: "packet serializer", serializes the 16-bit packet by splitting it up into two packets of 8 bits. - `pkt_des`: "packet deserializer", combines incoming 8-bit data over two cycles to one 16-bit packet. - `direction_decider`: Given some incoming packet, determine to which direction it should go. It uses a *very* simple decision procedure for this, defined in the if/else tree for `pick_packet`. If several packets arrive at once one of the packets is dropped.

Since every input and output port must be connected to something in Ex-PART, the `cap` component exists. It takes some PRU's output data and does nothing with it, and drives the input of that PRU with Nothing.

The layout of the manycore is done by defining a `pru` hierarchy. This system is chained four times in a vertical manner. Chain can only chain one input/output, so the remaining vertical connections in the other directions are defined manually. The horizontal connections are defined by using multi-connections. Constant drivers supply x and y coordinates to the routers, so they know where in the grid they are located. These coordinates are 1-indexed.

All around the perimeter of the grid caps are inserted, except at the west input of the top left component, and at the south output of the bottom right component. This input and output are connected to the I/O pins of the FPGA and are used to supply outside data to the system and retrieve data from the system.

For test runs the test input `mc_input` is provided, run it like this:

```
*Main> simulate @System manycore mc_input
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Just 69,Just 10,Just 69,Just 26,
```

This input sends two numbers to two different cores, which send their result to address 69, which is (4, 5) (`shiftL 4 4 + 5 = 69`), i.e. the bottom right, where the output is located, hence two packets have arrived at the output, the sums of the inputs.

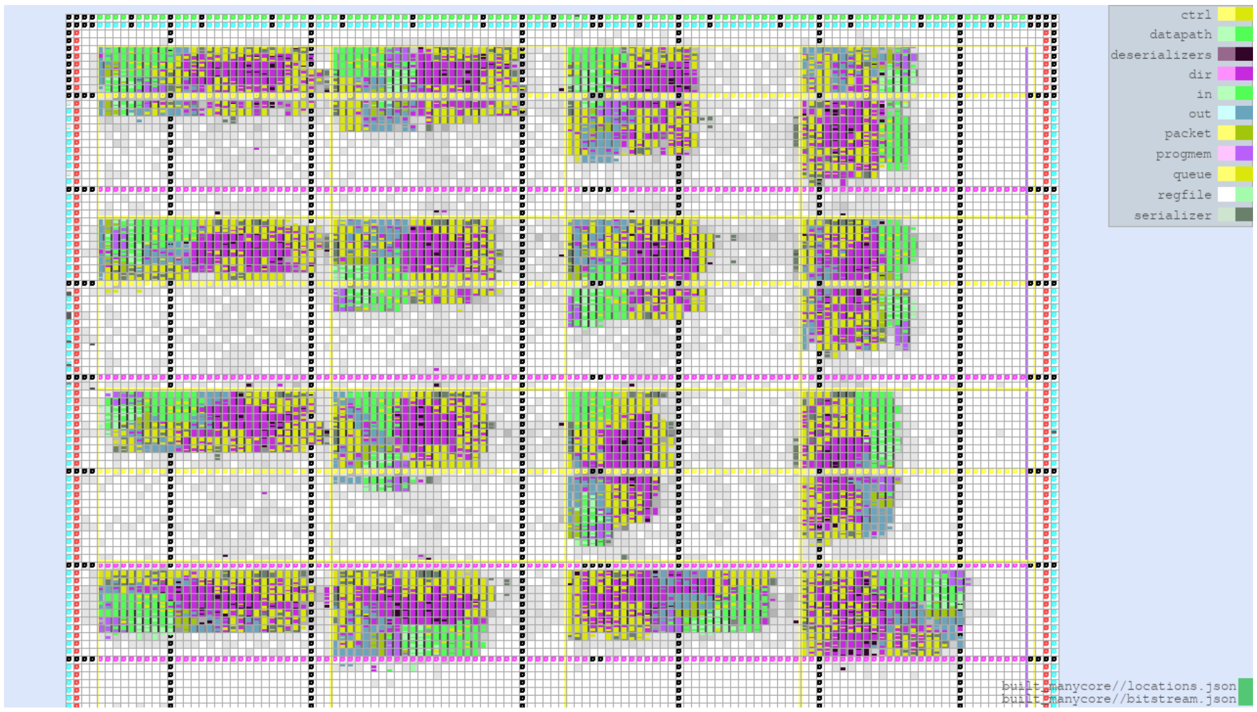

Figure 8.3: The manycore as seen in the visualizer

8.3 Feature Testing Examples

To test Ex-PART's features, examples where defined. This resulted in a nice library of examples, their purposes and workings are explained in this chapter.

chain

Demonstrates chain with a combinational summing structure. the `summer` component takes two inputs: a 6-bit partial sum and a 2-bit number to add to partial sum. Chaining the output of one instance to the the partial sum input of another creates a chain that sums all the inputs.

The summers need inputs, so the `sum_driver` provides these. It simply cycles through all values of an `Unsigned 2` over four cycles.

To show multi-connections and component accessors, the first component of the chain is driven by a constant value in line 20 of the `.expi`. A multi-connection connects the sum chain and the repetition of drivers.

```
*Main> L.take 10 $ simulate @System system []
[0,0,6,12,18,0,6,12,18,0 -- continues for ever
```

constants

Demonstrates constant drivers. A counter that must be enabled and has a configurable interval is driven by two constant values. The three most significant bits of the counter are sent to the output.

```
*Main> L.take 500 $ simulate @System system []
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

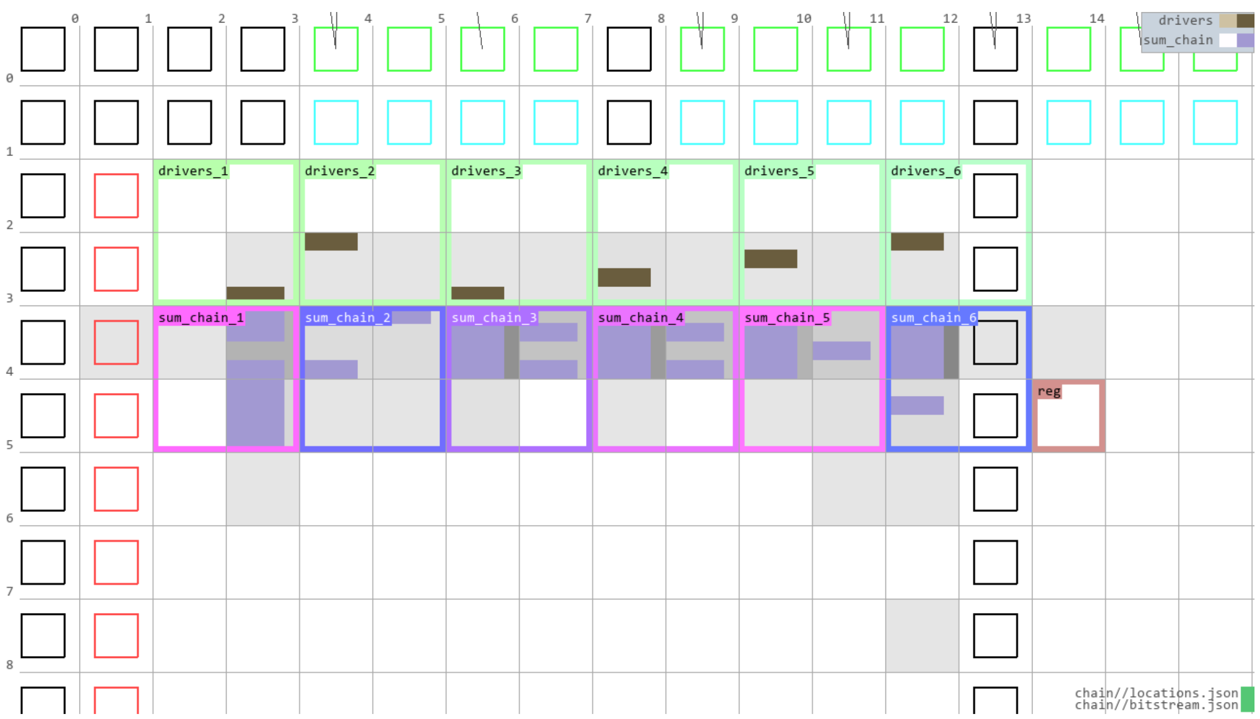

Figure 8.4: Chain as seen in the visualizer

```
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2
,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2
,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2
,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2
,2,2,2,2,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3 -- continues for ever
```

md5

Implementation of *one* MD5 hasher. This implementation was copied to build md5_reuse and md5_parallel. See the chapter on md5_reuse for more information on the MD5 hasher.

```
*Main> simulate @System system $ [Just 0, Just 0, Just 0, Just 0] L.++ (L.take 69 $ L.repeat Nothing)
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing
,Nothing,Just 869400926,Just 2891193117,Just 428080646,Just 1142714583
,*** Exception: X: finite list
```

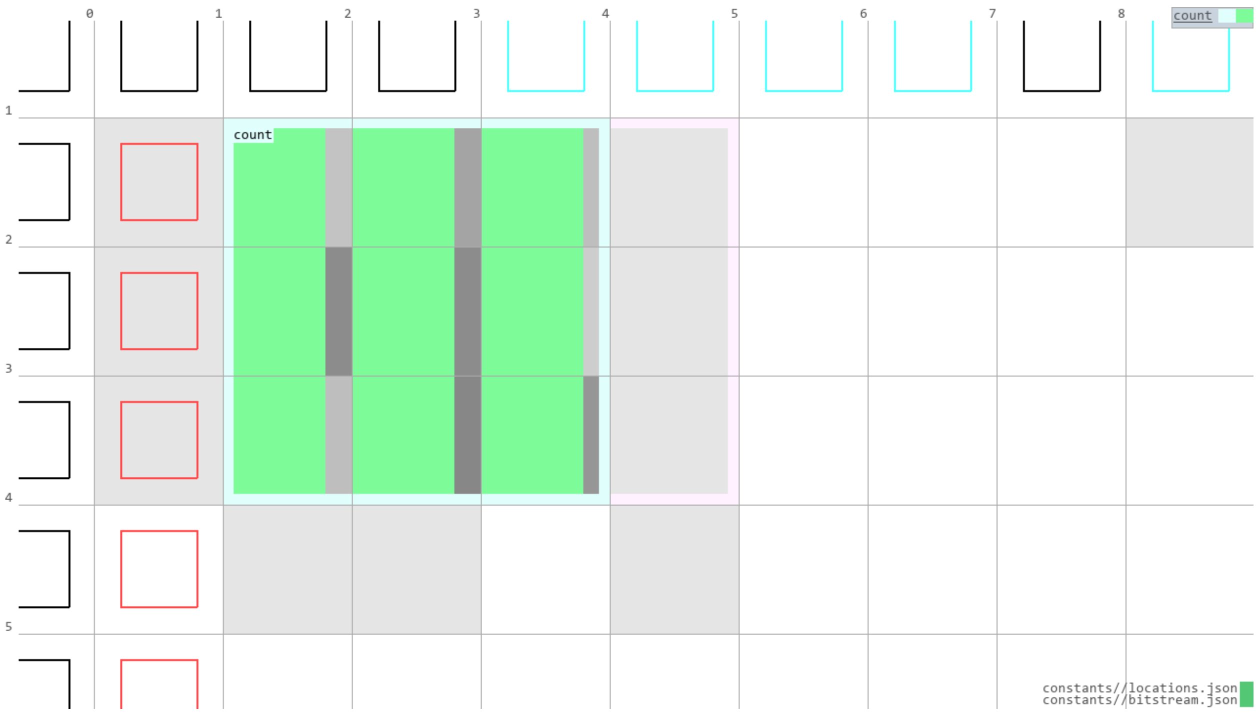

Figure 8.5: Constants as seen in the visualizer

Computing the hash of the 128 zeroes message on one hasher.

myhash

Demonstrates that systems can be used in chains and repeats. It implements a custom “hashing” algorithm: it applies a computation that’s inspired by MD5, just simplified very much and with unproven effectiveness. This was done because the MD5 system was quite unwieldy to work with, so using a smaller hashing algorithm is nicer to demonstrate everything with.

The chain at line 23 chains the system `calc_sys` defined in line 10-21.

```
*Main> simulate @System system [Just 5, Nothing]
[Nothing,Nothing,Nothing,Nothing,Nothing,Just 3090003402,*** Exception: X: finite list
```

pipe

A similar component as `pipeline` is used, but instead of it being chained it is instantiated three times and connected manually. This is a simple example that was used in a presentation to demonstrate Ex-PART’s main features.

```
*Main> simulate @System system [14,13,23,4]
[0,0,0,5,1,7,16,*** Exception: X: finite list
```

pipeline

Demonstrates a larger design than Collatz to examine how synthesis times are affected by Ex-PART’s strategies.

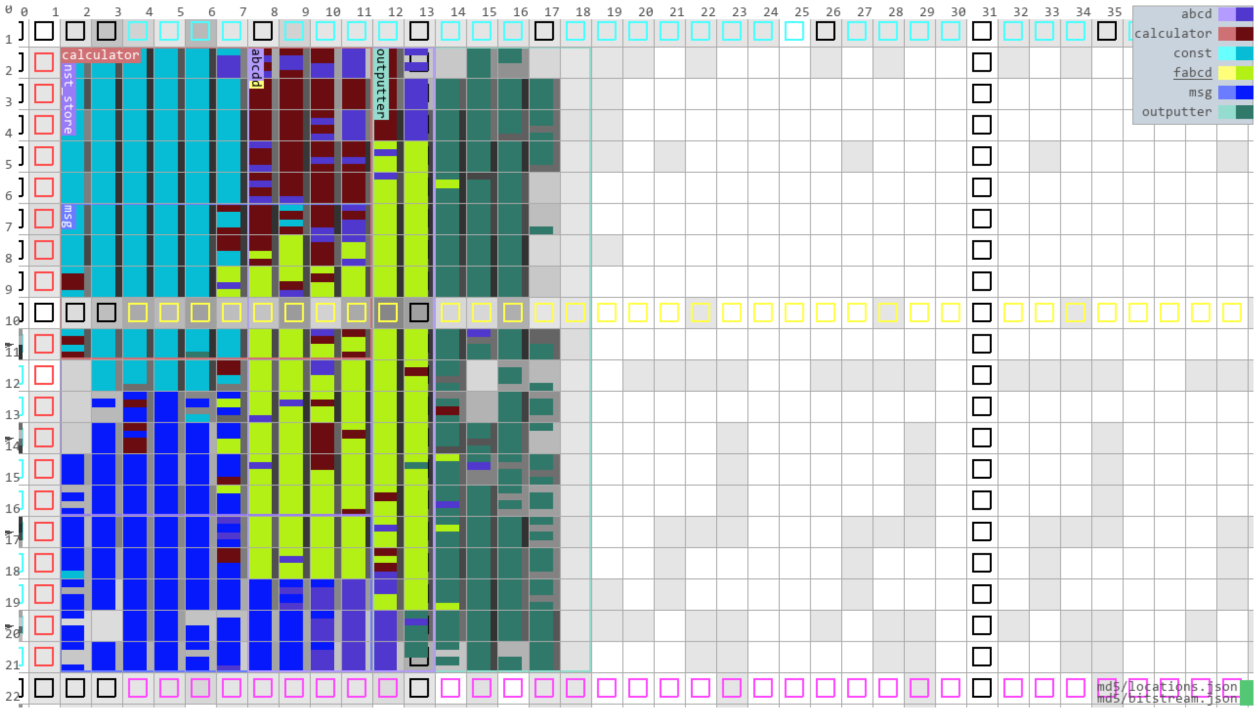

Figure 8.6: md5 as seen in the visualizer

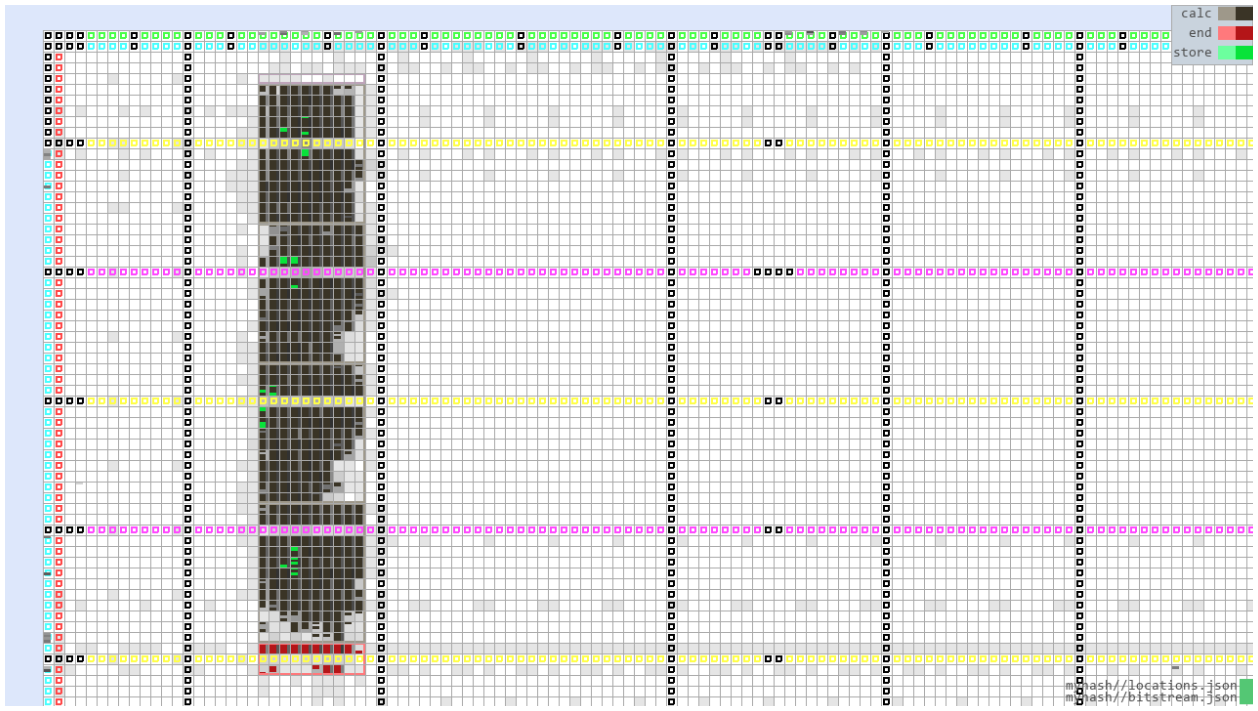

Figure 8.7: myhash as seen in the visualizer

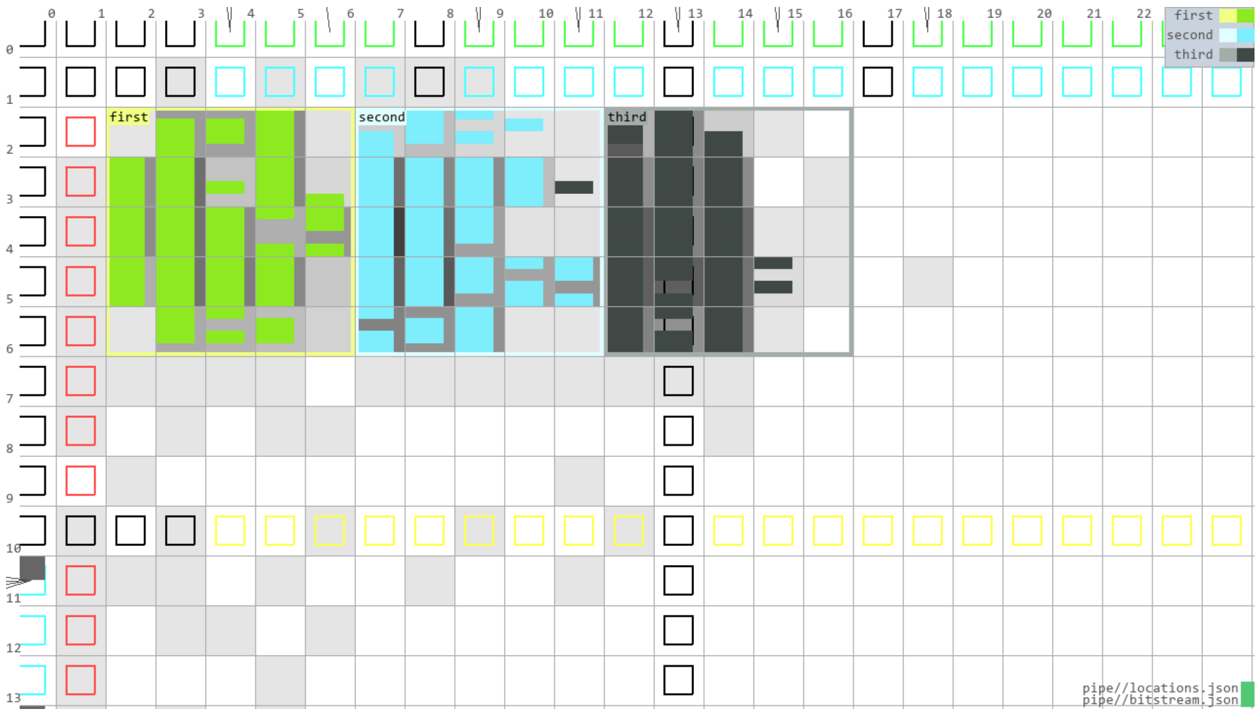

Figure 8.8: pipe as seen in the visualizer

Consists of one component, `compute`, that sets its state to its input, and applies a hardware-unfriendly computation to its state to compute its output:

```
 out = (s * 13) `mod` 17 - 4
```

Multiplication and modulo are very simple expressions, but they generate a lot of hardware.

Eight of these `compute` components are chained to form a simple pipeline. The computation is obviously not very relevant, this hardware design only intends to show how Ex-PART deals with a larger designs (this design was first defined a little after Collatz was built, hence it was one of the largest at the time).

```
*Main> simulate @System system [14,13,23,4]
[65532,65532,65532,65532,65532,65532,65532,65532,10,9,6,3,*** Exception: X: finite list
```

repeat

Demonstrates both repeat and several features of multi-connections.

The design describes 4 rows: 2 rows of counters and 2 rows enablers, the enablers decide whether counters continue, and the result of the first row of counters is used to enable the second row of counters. The second row of counters is wider than the first. Viewing this in the visualizer makes things most clear (see below example command)

The slightly irregular layout allows demonstrating both the :-multi-connections and ranges for multi-connections.

```
*Main> mapM_ print $ simulate @System system []
(0,0,0,0,0)
(1,1,0,0,0)
(2,2,1,1,1)
```

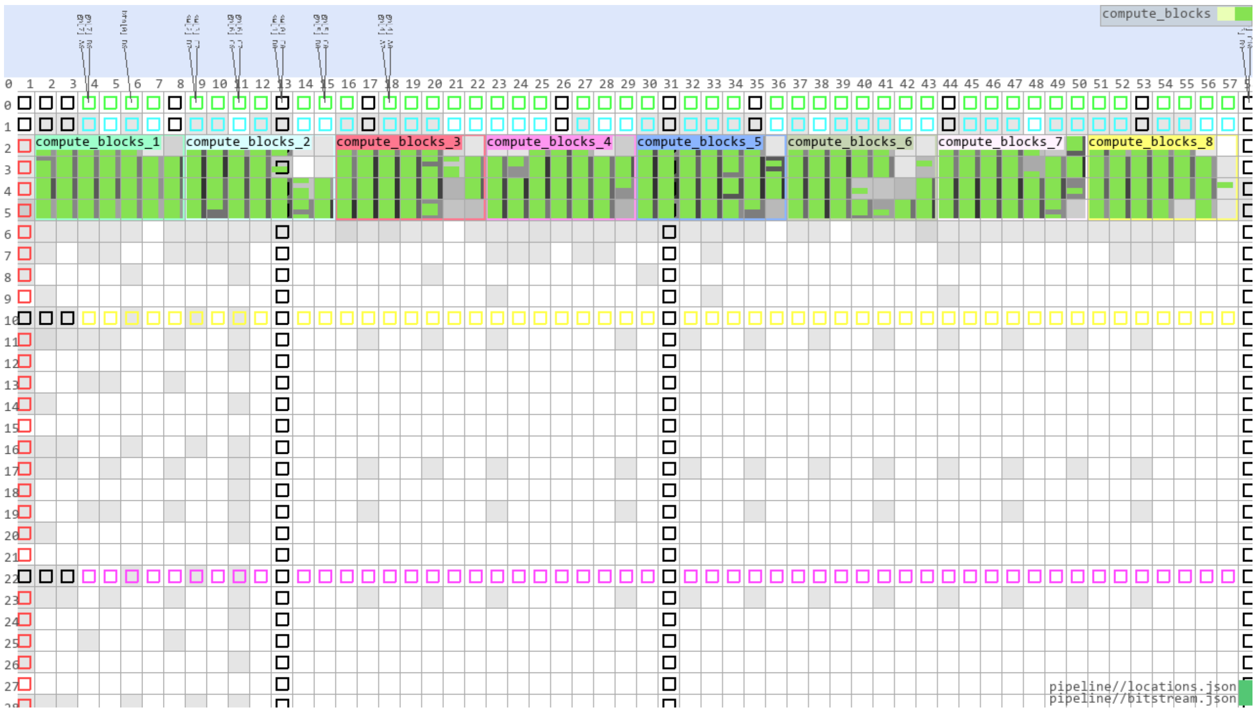

Figure 8.9: pipeline as seen in the visualizer

```
(3,3,1,1,1)
(4,4,2,2,2)
(5,5,2,2,2)
(6,6,3,3,3)
(7,7,3,3,3)
(8,8,4,4,4)
(9,9,4,4,4)
(10,10,5,5,5)
(11,11,5,5,5)
(12,12,6,6,6)
(13,13,6,6,6)
(14,14,7,7,7)
(15,15,7,7,7)
(0,0,8,8,8)
```

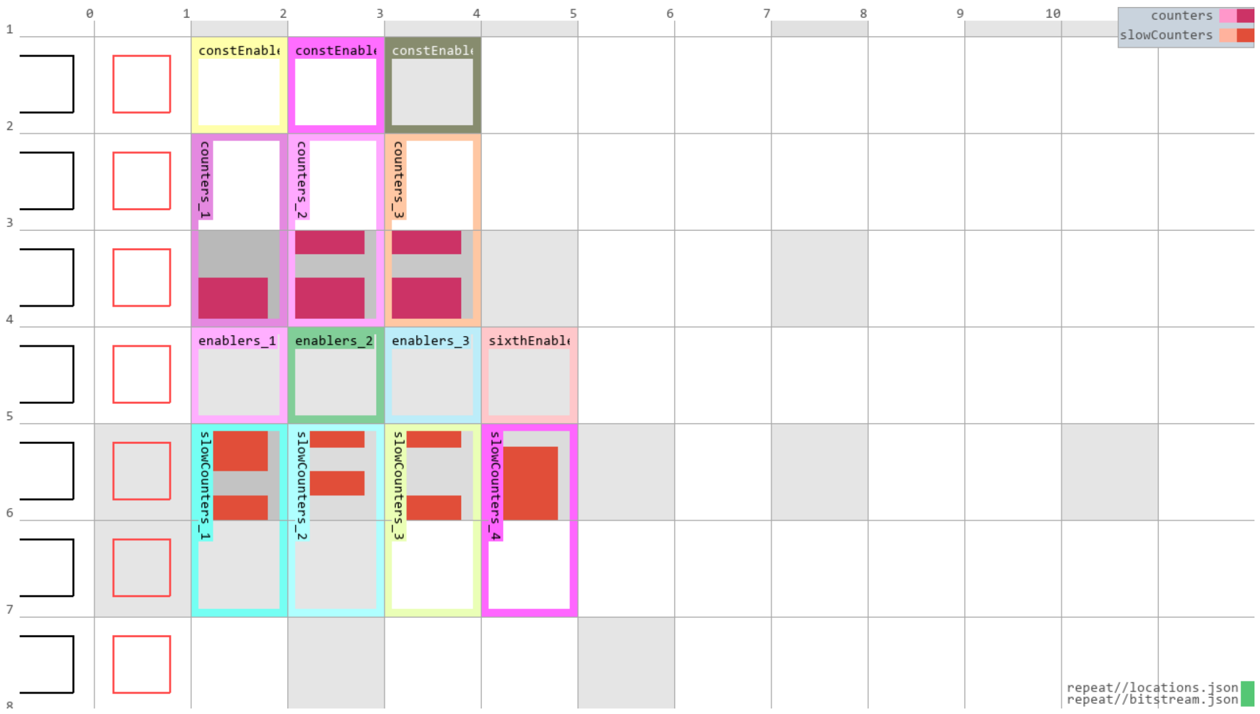

Figure 8.10: repeat as seen in the visualizer

Chapter 9

Maintenance

9.1 Introduction

This is the *maintenance manual* for Ex-PART. If you are planning on adding features or fixing bugs, you should read the relevant portions of this manual first. Information in this manual can also be of service when designing hardware and encountering unexpected or incorrect behavior (e.g. the error section).

9.2 Types

In `parser/Types.hs` types that are used in the entire project are defined. The idea behind most of them and their uses are outlined here. See also issue #18.

`LayoutExpr`

Recursive data type for containing size and coordinate expressions as they are defined by the designer. This data type is necessary as it is impossible to immediately *evaluate* expressions during parsing. With this data type expressions can be stored until they actually can be evaluated, after elaboration. There is an instance of the `Pretty` typeclass to pretty-print expressions to make them easier to debug.

`ISOStat` and `IOStat`

`ISOStat` is an `I`nput, `S`tate, or `O`utput statement. These contain the information defined in the statements at the start of a component definition. `IOStat` is an `I`nput or `O`utput statement, so the statements at the start of (sub-)system definitions. These types basically contain the same information, and perhaps merging them may be better, as very often it is necessary to do ad-hoc conversions between them. See also issue #21.

`Component`

Stores the information in a component definition: a list of `ISOStat`s, the name of the component type, and the where block. Currently there still is a list of constants arguments as well, but that is always an empty list. See also issue #19.

`Instance`

In instance is a laid out version of either a component or a system with a location and size on the FPGA. There are two data constructors for `Instance`: `CmpInstance` and `SysInstance`. The component instance constructor requires the actual `Component` object, such that that component does not have to be looked up every time when processing the instance. The system instance constructor just takes the name of the system it instantiates, and of course the rest of the parameters an instance needs.

`ExpcDesign`

The parse functions work towards parsing the `.expc` and `.expi` files to a `Design`, however as they need to be parsed in two separate steps this intermediate data type exists. It contains only the information that is present in an `.expc` file: a list of `haskell` blocks and a list of components.

`Design`

This data type contains the design as it is written down by the designer. It contains the same information as the `ExpcDesign`, but also a `SystemTree`, which stores the design described in the `.expi` file.

`SystemTree`

Represents the `.expi` file. It is a tree because one of its fields is of type `SystemTree`, hence it is a recursive datatype. This tree structure represents the hierarchy of systems that is defined in the `.expi`. `SystemTree` contains a lot of fields as there are many constructs available in an `.expi` file that later need to be elaborated.

`System`

The elaborated version of the `Design`. Elaboration is indeed simply a function `Design -> System`. `System` is also a recursive datatype, but its recursion is a bit more hidden. A `System` has:
- A name.
- A type, which is equal to the name, except for system instantiations and systems re-instantiated through chains and repeats.
- "TopData", which is only available in the top system, and contains the information from the `.expc` file.
- Size and coordinates.
- I/O statements.
- Connections.
- Constant drivers.
- Elements, separated in two lists, `elems` and `allElems`, where `elems` is exactly the elements that have a location. The recursion occurs here, as an Element can be another System.

`Element`

An element is an abstraction over components and subsystems, and contains the information where those types overlap, so an element has a name, type, size, coordinates, and I/O definitions. Furthermore, an element has an implementation, which is either the system or the component. Using the implementation it is still possible to differentiate between component and system wherever they need a different treatment.

`Connection(')` and `CID`

A `CID` is a combination of an element name and a port name. Two `CID`s can form a `Connection`. A connection is always ordered as `Connection from to`, so even if the arrow points to the other direction, the parser constructs the `Connection` such that this holds. `Connection'` is a connection including the bit width of the two ports that are connected. This is useful information to have during post-processing, and is determined during elaboration.

`RawRepetition`

A repetition as written down by the designer. This can be either a chain or a repeat. It has a name and coordinates and a *list of options*, like the amount of repetitions and layout procedure. Since the parser does not need to enforce whether all the options are present, some options may be missing in this list. That is why during elaboration this `RawRepetition` is converted to a `Repetition`, and throws an error when options are missing.

`Repetition`

An exact representation of a repetition like chain and repeat: in this type exactly every option is guaranteed to be present. During the construction of a value for `Repetition` errors could be thrown if any its values have not been specified by the designer.

`MultiConnection` and `MCID`

Similar to `Connection` and `CID`, but for multi-connections. An `MCID` also has an element name and port name, but additionally has a range, which is either `All` or a range from some integer to some other integer.

A multi-connection is, just like `Connection`, two `MCID` ordered as the originating `MCID` first, and the destination `MCID` second.

`ConstantDriver`

Similar to a `Connection`, but instead of an originating `CID`, the constant it drives is stored as a `String`. This string can later be converted to a series of bits in `Postprocessing.hs` to set the constant value to the correct ports.

9.3 Program Structure

This section walks through the directories of the source code in approximately the order that the program operates. For each file a description of what kind of functions are located there and what everything is supposed to do is present.

Compiler (`compiler/`)

Contains all the compilation flow definitions and a lot of helper functions to construct these flows.

`Compiler.hs`

Contains the actual compilation flows of type `Flow` that are described in setting up. If you want to add a new flow, build a function of type `Flow` here. Its helper functions should be in `Flows.hs` and `Steps.hs`.

`Flows.hs`

Contains functions that are pretty much only a monadic concatenation of `IO ()` actions. Organized as such, the flows read very much like a script. Flows defined here any arguments, using those they concatenate steps from `Steps.hs`.

`Steps.hs`

Contains many `IO ()` actions that can be concatenated in a flow in `Flows.hs`. Each of these steps is accompanied by a neat `putStrLn` that prints what step is being taken. This way any flow prints the steps it is currently taking, informing the user of what is happening. This is really nice to have as some steps can take quite a long time, and it is nice to see that *something* is happening. Some steps (like `compileToVerilog`) print intermediate messages as well.

Parser (`parser/`)

Just as the first step of any compiler, first the source files must be parsed. In the `parser` directory all the Parsec functions necessary to parse both the component file and the instantiation file are located.

The type definitions used basically everywhere in the program are defined in a source file here as well. This could be placed in a more logical position at some point (issue #18)

`Parse_shared.hs`

Parser definitions shared between the `.expi` and `.expc` parser. The very hacky Haskell parsing mentioned in issue #1 is located here, mostly in `haskell_type` and `haskell_stat`.

This also defines some standard often used parsers like `whiteSpace` and `parens`.

`Parse_expc.hs`

`expcdesign` can parse one entire `.expc` file. As the order of statements mostly does not matter in Ex-PART, the data structures just keep lists of *types* of statements. That's why in `expcdesign` there is a `sorter`, which groups statements according to types.

Furthermore, `isoStatement` parses **i**nput, **s**tate, and **o**utput statements. When no initial state is given, quite an unclear error message pops up. Usually if a parse error occurs in this function, it's because of a missing initial state.

`Parse_expi.hs`

`system` parses the `.expi` file. It uses a similar strategy as in `Parse_expc` of parsing to a list of `Statement`s, and then sorting those such that it fits nicely in the `SystemTree` datatype.

The parsers here are pretty elaborate, this is because they must be able to parse all the features that are later elaborated (like chains and repeats).

`Parser.hs`

Contains three parsers, that operate on file paths. These make it convenient to parse the two files. `parse_both` combines the two parsers to parse two files to one `Design`.

`Types.hs`

Types defined here are explained in the Types section.

Besides just important types used in the project, the function mapping Haskell types to bit widths is also located here. Regarding this function, see issue #2.

Elaboration (`elaboration/`)

Parsing parses to a ***`Design`***, while Ex-PART operates on a ***`System`***. Elaboration makes the conversion. It unrolls chains, repeats, and multi-connections, and elaborates system instantiations.

To better understand how elaboration works, compare your `.expi` file with `elaborated.expi` in the output directory. `elaborated.expi` is a pretty printed version of the `System` that your `Design` was elaborated to. In this file the comments after a connection is the bit width of that connection.

`Elaboration.hs`

`elaborate` applies all the elaboration steps on a `Design` to obtain a `System`. This is more of a wrapper, as the interesting recursive elaboration function is `elaborateSystem`. That function creates a `System` record and fills in all the properties. As some of these properties contain more systems (some of which are again copies of other subsystems), this function builds up the system recursively by going through the `SystemTree`. The properties are obtained from the given `SystemTree` and `Design` on which it operates, and on the results obtained by functions in the other files in this directory.

`Repetition.hs`

Supplies functions to unroll both kinds of repetition. It converts a `RawReptition` to a `Repetition`, which fits the list of options that the user supplied to a record containing exactly the necessary information. Once fitted, the `Repetition` is unrolled: for the elements, it generates $amount new elements (i.e. components or systems) and appends those to the system. For the chain, connections are also generated and appended to the connection list.

`ElaborateConnection.hs`

To make Yosys post-processing much easier, we already calculate the bit width of every port here. `Connection` does not have bit width, `Connection'` does. `elaborateConnection` finds the bit width for every connection. It's good to do this this early in the process as well, since many common errors regarding connections are caught by this step and that saves waiting on Clash and Yosys before discovering these.

`Multiconnection.hs`

In a similar fashion to repeats and chains, multi-connections need to be unrolled. A multi-connection connecting n ports is simply converted to n connections connecting one port of the element with the correct name.

JSON Builder (`json-builder/`)

After elaboration the system is ready to be built. The first step in the Ex-PART build process is to generate a `locations.json`, which is later used to constrain LUTs to the correct area on the FPGA.

`Locations.hs`

Four important functions are defined here. `allInstsWithCoords` restructures the data to a new type that the rest of the file can work with. This type is a three tuple, which does not help in clarity of the code. A nicer type could be designed for this.

`hasCycle` checks whether there is a cyclic dependency in coordinate or size definitions. See also issue #13.

`reduceAll` reduces expressions, i.e. given an expression with variables, and a mapping of variables to values, it reduces that expression to (hopefully) a constant. It uses a recursive approach for this evaluation, reducing any other expression it encounters.

`relToAbs` takes reduced expressions, which are still relative to their parent systems, and turns them into absolute coordinates on the FPGA by adding the parent's coordinates.

`JSONBuilder.hs`

Converts the data in a `System` to something that `Locations.hs` can work with, then by combining the functions in `Locations.hs` obtains a tree of absolute positions. This is exactly what nextpnr allows for constraining, so that can be written to `locations.json`.

Clash Generator (`clash-generator/`)

To be able to synthesize the components, they are converted to Clash. In this directory there is also code for creating simulation files: Clash files representing the entire design to verify functional correctness.

`Generator.hs`

Wrapping functions using the `doPreliminaryProcessing` function from `Preliminary.hs` and `toClash` from `ComponentConversion` to generate Clash code for the components, and using The Flattener module to define a default `flatten` function.

`Preliminary.hs`

Creates the `builds/` directory in which all the components will have their own directory, in which it creates directories for each component with the name of the component as the name of the directory. It also creates `Definitions.hs`, this is the file containing all `haskell` blocks of the component file. To `Definitions.hs` some preamble is added in the `genDefs` function. If you need any GHC extensions or imports enabled during simulation, this is where you could add them.

`ComponentConversion.hs`

Converts components to Clash. For each component a function is generated, this function has the type that Clash's `mealy` function expects: `s -> i -> (s, o)`. The `toClash` function combines every step. In summary:

1. The type signature is generated from the information in the I/S/O statements.
2. The equation is generated: the function name, state argument, input argument, and the two-tuple that a mealy machine emits in Clash is generated.
3. The where statement for this component is generated by simply copying the transition expressions, and any other expressions the designer may have specified to the where clause.
4. A `topEntity` is defined, and decorated with synthesis annotations to ensure it synthesizes in a predictable way.

To gain more insight into what exactly this file generates, take a look at an output file looking like `builds/component/Synth_component.hs` in the output directory of your project. The Clash code produced is usually quite readable.

`Flattener.hs`

Generates simulation files by ‘flattening’ the design: Starting at the top level system, it creates a Clash function where the connections and instantiations are defined exactly as in the `.expi`. Then for any subsystem it creates additional functions, recursively.

To use mealy machines defined in the `.expc` on the `Signal` level, it generates for every mealy machine with name e.g. `component` a function `componentM` that operates on the `Signal` level.

Constant drivers are generated by creating an extra statement in the where clause like `const_0 = pure 0`, for a constant driver of 0.

A topEntity is also defined, so that the system can be converted to Verilog as one system, instead of just the components. The monolithic and hierarchic flow use this topEntity.

Yosys (`yosys/`)

Takes care of much of the Yosys-related stuff: running the tool and applying necessary pre- and post-processing steps. The organization of this module is a little weird: some Clash related stuff, like compiling generated Clash to Verilog also happens here (issue #22)

`Preprocessing.hs`

To make synthesis much faster, its better to concatenate all the loose Verilog files generated by Clash, and synthesizing that. However, if the top module does not use some module, then Yosys will not synthesize that (as an obvious optimization). To circumvent this a dummy top module is generated, that simply instantiates every component once, and routes inputs and outputs directly to and from each instance from the arguments of the top. This top module can later be deleted, and our own hierarchy of modules can be inserted. This happens in `Postprocessing.hs`.

Here definitions for Clash processes are generated as well for each component. The function `proc` from `System.Process` can simply be monadic mapped (`mapM`) over a list of these definitions to run all the processes.

`Yosys.hs`

Contains functions to compile the Clash file for all components to Verilog, and calls the pre- and post-processing steps. Also contains synthesis functions for the hierarchic and monolithic functions, and of course an `IO ()` action to actually run Yosys to synthesize generated Verilog

`Postprocessing.hs`

Generates a JSON (`interconnect.json`) representing the connections and instantiations in the instantiation file. Data structures representing Yosys' JSON structure are defined first, and Aeson `ToJSON` instances are defined for these. Then, given a `System` a list of `Module`s is produced. This process is very non-trivial and very sensitive to bugs. The main problem is that Yosys defines their connections via unique integers inside a module, and Ex-PART's are defined through module and port names. Therefore a map from module and port names to these integers must be created first, and then everything needs to be arranged exactly as Yosys does in the JSON file.

This JSON goes (almost) directly to nextpnr to be placed and routed. Bugs in this part of the project, like incorrectly connecting bits in a bus, cannot be simulated anymore and are very hard to find. Manual verification of the process has occurred for the Collatz example, but not for any larger designs.

Other Files

This directory contains several `.ys` files, these are scripts for Yosys to run. `grouped.ys` is for the default flow, `monolithic.ys` and `hierarchic.ys` should be self-explanatory.

`merge_json.py` merges the Yosys JSON with just component modules and the JSON generated by Postprocessing. This is done by a Python script instead of with Aeson because it was a little bit more convenient at the time... It should of course just be done with Aeson, as that is much neater (issue #24).

Nextpnr (`nextpnr/`)

Runs nextpnr on the combined JSON files, `synthesized.json`.

`Nextpnr.hs`

Only contains one function, `nextpnr`, that runs `nextpnr` for the ECP5 with 85k LUTs. If any of the settings for nextpnr turn out to not fit your use-case, modify them here. Note that functions calling nextpnr (e.g. in `Steps.hs` or `Flows.hs`) may provide extra options for the process.

Just as every other tool, nextpnrs output and error streams are logged to a file.

`constrainer.py`

Nextpnr allows scripts to be run just before certain steps in placement and routing. In `Nextpnr.hs` this python script is set to be run just before placement. Any Python script nextpnr runs has access to the `ctx` object that contains all the cells to be placed, and nets to be routed. This script goes through all the cells, and based on the name of the cell, looks up where in the JSON the rectangle constraining the cell should be, finds that rectangle, and constrains the cell to that rectangle. Not every cell will have a standard name, as for example I/O cells are not part of the Ex-PART specification. These cells are therefore not constrained.

Visualizer (`visualizer/`)

While technically not part of the program Ex-PART, the visualizer is quite an important part of the workflow. It is implemented in Python, using Pygame as a graphics library.

`color.py`

A small library implementing some commonly used color features. As slices and much other stuff is colored based on its name, and colors are randomizable, this is all handled centrally here.

`init.py`

Contains everything that many modules might need. Some global zooming and viewing variables, argument parsing, Pygame initialization, and a function handling (keyboard, mouse) events.

`files.py`

Function for monitoring and reloading `bitstream.json` and `locations.json`, the two files that the visualizer can show. There is also the drawing function for drawing indicators in the bottom right of the screen showing which file is loaded.

`iodb.json`, `blinky.lpf` and `parse_iodb.py`

To render the IO on the sides of the FPGA, Trellis' IO database (`iodb.json`) had to be parsed and linked to names for the I/O pins. This `blinky.lpf` contains the default names of many sites of the I/O pins, and `parse_iodb.py` is a script that can parse the I/O DB, link it to names in the `lpf`, and generate an easy to visualize CSV. as long as `iodata.csv` is available and valid you probably don't need to touch these.

`iodata.csv` and `tiledata.csv`

Some features of the ECP5 are described in these CSVs. `iodata.csv` links locations to pin names, so its easy to know exactly where a pin is on the FPGA. This helps in placing the design as it enables the designer to place the I/O of the design near the actual I/O pins.

`tiledata.csv` contains the tiles that are *not* LUT tiles. These tiles are also visualized in the tile grid as differently colored squares. The data for this file comes from here.

`grid.py`

Functions for drawing the grid, special tiles, and I/O pins.

`slice.py`

Functions for drawing slices. Slices are drawn with a random color if their name abides the Ex-PART naming conventions, otherwise they are drawn in a random shade of gray.

`routing.py`

Functions for showing usage of routing resources, by drawing darker shades of gray on tiles if more resources are used. It draws this based on a "routemap" which is regenerated if the bitstream has been updated.

`legend.py`

Functions for drawing a legend which shows which color is used for which module.

`connections.py`

Off by default, as computing this costs a long time for even slightly larger designs. Can be enabled with `-c`. Functions for drawing *outgoing* connections of a component. This is a neat way to visualize *where* the outputs of a component are located, and where the data comes in. Certainly for large components this can give insights into why placement and routing may be difficult.

`systems.py`

Functions for drawing boxes as defined in the `locations.json`. Goes through the JSON file recursively and draws boxes for every bottom level component it finds at the specified location.

`main.py`

Contains the main loop which calls all the functions defined in the other files for drawing, event handling, file loading, and view updating.

9.4 Feature Implementations

Below the same feature list as in the programming manual is shown. Here a brief explanation of the implementation of the feature is provided, including where to find the code.

Comments

Technically partially implemented in GHC, as comments near transition expressions are simply copied to Haskell. For `.expi` files, Parsec does the heavy lifting. By defining a `lexer` in `Parse_shared.hs` with `haskellDef` as language definition, haskell style comments are automatically supported as white space. After parsing the comments have been ignored, as is the intention.

`haskell` block

`haskell` blocks are stored in the `Design` as a list of `HaskellDefs`. This list extracted by `genDefs` in `Preliminary.hs` to generate `Definitions.hs`. This file is then imported by every other Clash file generated by Ex-PART such that the definitions can be used.

Component definition

In `ComponentConversion.hs` most of the processing on components happens. Here they are all converted to Clash files as described `ComponentConversion.hs`.

Coordinates and Sizes

All coordinates and sizes are stored as `LayoutExprs`. In `Location.hs` the coordinates and sizes are processed into the `locations.json` file.

System Definitions

System definitions are parsed in `Parse_expi.hs` and are then elaborated in `Elaboration.hs`. The size and position of a system are enforced in `Nextpnr.hs` and `constrainer.py`. Connections and instances are made in `Postprocessing.hs` for the ECP5, and in `Flattener.hs` for simulation.

Component Instantiation

Instantiations before placement are done by instantiating cells of the component name in the Yosys JSON in `Postprocessing.hs`. The instantiation location and size are enforced in `Nextpnr.hs` and `constrainer.py`.

Port connection

Port connections are made in `Postprocessing.hs` by generating a `NetMap = Map CID Net`, where a `Net` is simply an integer: specifically, the unique integers Yosys uses to denote which ports are connected. For more information on Yosys' connection system, view the Yosys manual (PDF), in appendix C.218. The `NetMap` thus assigns an integer to ports, the same integer is assigned to two ports if they are connected via a `Connection`.

Constant Drivers

Constant drivers are implemented in `Postprocessing.hs`, there a Yosys JSON module is generated for a constant driver, which is instantiated in the correct place to implement the design. For simulation lines like `const_0 = pure 0` are added to where statements of systems containing constant drivers in `Clash.hs`.

Repeat and Chain statement

Repeat and chain statements are fully implemented during elaboration as they are simply unrolled to simpler elements.

Multiconnections

Multi-connections are unrolled to simple single connections during elaboration. From then on the generated connections are indistinguishable from regular connections.

Unplaced Systems

A `System` has two fields for elements, one which contains *all* elements, including the unplaced ones, and one which contains just the elements which need to be put somewhere. When instantiating a system the field with all elements is filtered on the name of the system that need to be placed, and when `Postprocessing.hs` actually instantiates all the systems only the field with the placed elements are instantiated.

System Instantiation

During elaboration, if a system instantiation is encountered, the system is simply copied, but with a different name.

9.5 Error List

This lists errors that may be thrown that are not listed in the programming manual. If you see any of these errors, there is most likely a bug in Ex-PART, and not in your code. The information here may point you to where to search for a solution.

Errors

- `clash-generator/ComponentConversion.hs:82:` A state is not a port.
    - During component conversion, an `ISOStatement` constructed as an `SState` was given to the function `toPortName`. This function is a local function, and is called only on filtered lists of `ISOStatement`s such that either only inputs or only outputs are passed, hence this error should never appear.
- `clash-generator/Preliminary.hs:38:` Something went wrong during elaboration, the top system does not have top-data.
    - Elaboration generates a `System`, which has a property `topdata`. The idea is that the root node of the `System` hierarchy has the `topdata` set to the information in the expc file. If this hasn't happened, something is broken in `Elaboration.hs`.
- `compiler/Compiler.hs:100:` How can there be several components with the same name?
    - This code is part of the `auto` flow, which is a little bit broken. No research has been done where exactly it breaks as it is quite time-consuming and hard to reproduce (issue #7).
- `json-builder/Locations.hs:101:` Coordinate reduction found non-constant value ($expr)
    - The reduction should only be run on constant coordinate expressions, so any width/height/x/y variable must have been substituted by a constant value. Probably the error "Could not find ID in provided list" will be thrown earlier than this one, as that appears in a function that turns expressions into expressions without variables, and that function is always called before the function this error appears in.
- `nextpnr/Nextpnr.hs:28:` nextpnr terminated with code $code
    - This error may occur when assertions in nextpnr are tripped, and these can be tripped by incorrectly constraining LUTs, e.g. to a "negative" area rectangle (by setting reversing the top left and bottom right). It may also be the case that something weird was defined in the JSON generated by `Postprocessing.hs`.
- `parser/Types.hs:102:` Invalid ISOStatement for bitwidth (state not implemented)
    - In the current implementation of Ex-PART, the bit width of states is irrelevant. Therefore no conversion to bit width from states is implemented. This function *probably* is never called with a state, but the types cannot guarantee this right now.
- `yosys/Preprocessing.hs:57:` No state should have been seen here.
    - Again the issue with ISOStatements appearing where only I/O is relevant... (issue #21) This function is again only called with filtered statement lists, so this error should never trip. These next two errors are to catch the same problem, just more specifically for either only inputs or only outputs.
    - `yosys/Preprocessing.hs:68:` Not an input: $x
    - `yosys/Preprocessing.hs:74:` Not an output: $x

Bibliography

[1] P. J. Staal, H. H. Folmer, "Computational complexities in modern FPGA synthesis flows," *unpublished*, 2022.

[2] L. Vega, J. McMahan, A. Sampson, D. Grossman, and L. Ceze, "Reticle: a virtual machine for programming modern FPGAs," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 756–771. [Online]. Available: https://doi.org/10.1145/3453483.3454075

[3] E. Axelsson, K. Claessen, and M. Sheeran, "Wired: Wire-aware circuit design," vol. 3725, pp. 5–19, 10 2005.

[4] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. MilanoviÄĞ, "Yosys+nextpnr: an open source framework from Verilog to bitstream for commercial FPGAs," 2019.

[5] C. Wolf, "Yosys manual." [Online]. Available: http://www.clifford.at/yosys/files/yosys_manual.pdf

[6] "Ex-PART Issues," https://github.com/PietPtr/Ex-PART/issues, accessed: 2022-02-14.

[7] P. J. Staal and H. H. Folmer, "Ex-PART." [Online]. Available: https://github.com/PietPtr/Ex-PART

[8] "Trellis DB," http://yosyshq.net/prjtrellis-db/, accessed: 2022-02-10.

[9] "Clash FIFO," https://github.com/clash-lang/clash-compiler/blob/master/examples/Fifo.hs, accessed: 2022-02-10.

Appendices

Appendix A

Case Studies Code

A.1 Collatz Conjecture Calculator

collatz.expc

```
haskell {
(>>>) :: Bits a => a -> Int -> a
(>>>) = shiftR

(<<<) :: Bits a => a -> Int -> a
(<<<) = shiftL

type Value = Unsigned 16

}

component router() {
    input val : Value
    output odd : Maybe Value
    output even : Maybe Value

    even = if testBit val 0 then Nothing else Just val
    odd  = if testBit val 0 then Just val else Nothing
}

component onEven() {
    input val : Maybe Value
    output res : Maybe Value

    res = case val of
        Just v -> Just $ v >>> 1
        Nothing -> Nothing
}

component onOdd() {
    input val : Maybe Value
    output res : Maybe Value

    res = case val of
        Just v -> Just $ (v <<< 1 + v) + 1
        Nothing -> Nothing
}

component merger() {
    input vo : Maybe Value
    input ve : Maybe Value
    output res : Value

    res = case vo of
        Just v -> v
        Nothing -> case ve of
            Just v -> v
            Nothing -> 0
}

component control() {
    input next_val : Value
    input set_val : Maybe Value
    state last_val = 0 : Value
    output result_value : Value

    last_val' = case set_val of
        Just new_value -> new_value
        Nothing -> next_val

    result_value = last_val
}
```

collatz.expi

```
system in (6, 6) at (2, 2) {
    input setting : Maybe Value
    output result : Value

    controller is control in (6, 1) at (0, 0)

    collatzer.val_in<-controller.result_value
    collatzer.val_out->controller.next_val

    controller.set_val<-setting
    controller.result_value->result

    collatzer in (controller.w, 4) at (0, controller.h) {
        input val_in : Value
        output val_out : Value

        router is router in (1, onOdd.h + onEven.h) at (0, 0)
        onOdd is onOdd in (collatzer.w - 2, 2) at (collatzer.x + 1, 0)
        onEven is onEven in (onOdd.w, onOdd.h) at (onOdd.x, onOdd.h)
        merger is merger in (1, onOdd.h + onEven.h) at (onOdd.x + onOdd.w, 0)

        router.val<-val_in
        router.odd->onOdd.val
        router.even->onEven.val
        onOdd.res->merger.vo
        onEven.res->merger.ve
        merger.res->val_out
    }
}
```

A.2 Parallel MD5 Hasher

md5_reuse.expc

```
haskell {

import Data.List as L

start :: Hash
start = (0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476)

zero_hash :: Hash
zero_hash = (0, 0, 0, 0)

start_hash :: Hash
start_hash = (1732584193,4023233417,2562383102,271733878)

-- a message of four zeroes
zero_msg :: Vec 16 UInt
zero_msg = (0:>0:>0:>0:>128:>0:>0:>0:>0:>0:>0:>0:>0:>0:>0:>4:>Nil)

hash_a (a,_,_,_) = a
hash_b (_,b,_,_) = b
hash_c (_,_,c,_) = c
hash_d (_,_,_,d) = d

data MD5State = Loading (Unsigned 2) | Hashing (Unsigned 6) deriving (Show, Generic, NFDataX)

start_state = Loading 0

add_hash :: Hash -> Hash -> Hash
add_hash (a,b,c,d) (a',b',c',d') = (a+a', b+b', c+c', d+d')

data_ctr_state = LoadingData 0

data DataCtrState = Ready | LoadingData (Unsigned 2) deriving (Show, Generic, NFDataX)

type UInt = Unsigned 32
type Hash = (UInt, UInt, UInt, UInt)

}
```

```
component const_store() {
    input stall : Bool
    state i = start_state : MD5State
    output k : UInt
    output s : Unsigned 5
    output out_i : Unsigned 6
    output is_hashing : Bool
    output ready : Bool

    ready = case i of
        (Loading _) -> False
        (Hashing n) -> n == 63

    k = k_store !! index
    s = s_store !! index

    -- We assume that loading will actually happen in those four cycles, and will happily start
    -- hashing even if that didn't happen.
    i' = if stall then i else case i of
        Loading 3 -> Hashing 0
        Loading n -> Loading (n + 1)
        Hashing 63 -> Loading 0
        Hashing n -> Hashing (n + 1)

    out_i = index
    load_i = case i of
        Loading step -> step
        Hashing _ -> 0

    is_hashing = case i of
        Loading _ -> False
        Hashing _ -> True && (not stall) -- Ja dit kan korter

    index = case i of
        Loading _ -> 0
        Hashing step -> step

    k_store =
        3614090360:>3905402710:>606105819:>3250441966:>4118548399:>1200080426:>2821735955:>
        4249261313:>1770035416:>2336552879:>4294925233:>2304563134:>1804603682:>4254626195:>
        2792965006:>1236535329:>4129170786:>3225465664:>643717713:>3921069994:>3593408605:>
        38016083:>3634488961:>3889429448:>568446438:>3275163606:>4107603335:>1163531501:>
        2850285829:>4243563512:>1735328473:>2368359562:>4294588738:>2272392833:>1839030562:>
        4259657740:>2763975236:>1272893353:>4139469664:>3200236656:>681279174:>3936430074:>
        3572445317:>76029189:>3654602809:>3873151461:>530742520:>3299628645:>4096336452:>
        1126891415:>2878612391:>4237533241:>1700485571:>2399980690:>4293915773:>2240044497:>
        1873313359:>4264355552:>2734768916:>1309151649:>4149444226:>3174756917:>718787259:>
        3951481745:>Nil
    s_store =
        7:>12:>17:>22:>7:>12:>17:>22:>7:>12:>17:>22:>7:>12:>17:>22:>5:>9:>14:>20:>
        5:>9:>14:>20:>5:>9:>14:>20:>5:>9:>14:>20:>4:>11:>16:>23:>4:>11:>16:>23:>
        4:>11:>16:>23:>4:>11:>16:>23:>6:>10:>15:>21:>6:>10:>15:>21:>6:>10:>15:>21:>
        6:>10:>15:>21:>Nil
}

component message_store() {
    input g : Unsigned 4
    input data_in : Maybe UInt
    state data_ctr = data_ctr_state : DataCtrState
    state message = zero_msg : Vec 16 UInt
    output m : UInt
    output stall : Bool

    (message') = case data_in of
        (Just d) -> case data_ctr of
            (LoadingData ctr_index) -> (replace ctr_index d message)
            _ -> message
        Nothing -> (message)

    data_ctr' = case data_in of
        (Just d) -> case data_ctr of
            LoadingData 3 -> Ready
            LoadingData n -> LoadingData (n + 1)
        Nothing -> data_ctr

    stall = case data_ctr of
        Ready -> False
        _ -> True

    m = message !! g
}

component fabcd_update() {
    input f : UInt
    input mg : UInt
    input s : Unsigned 5
    input k : UInt

    input in_A : UInt
    input in_B : UInt
    input in_C : UInt
    input in_D : UInt

    output out_A : UInt
    output out_B : UInt
    output out_C : UInt
    output out_D : UInt

    f' = f + in_A + k + mg
    out_A = in_D
    out_D = in_C
    out_C = in_B
    out_B = in_B + rotateL f' (fromIntegral s) -- Kan dit wel, variabele rotate?
}

component calculator() {
    input a : UInt
    input b : UInt
    input c : UInt
    input d : UInt
    input i : Unsigned 6

    output out_F : UInt
    output out_g : Unsigned 4

    stage = i `shiftR` 4

    out_F = case stage of
        0 -> (b .&. c) .|. ((complement b) .&. d)
        1 -> (d .&. b) .|. ((complement d) .&. c)
        2 -> b `xor` c `xor` d
        3 -> c `xor` (b .|. (complement d))

    out_g = case stage of
        0 -> resize i
        1 -> resize (5 * i + 1)
        2 -> resize (3 * i + 5)
        3 -> resize (7 * i)
}

component abcd_store() {
    input enable : Bool
    input in_A : UInt
    input in_B : UInt
    input in_C : UInt
    input in_D : UInt

    state s_A = 1732584193 : UInt
    state s_B = 4023233417 : UInt
    state s_C = 2562383102 : UInt
    state s_D = 271733878  : UInt

    output out_A : UInt
    output out_B : UInt
    output out_C : UInt
    output out_D : UInt

    (s_A', s_B', s_C', s_D') = if enable
        then (in_A, in_B, in_C, in_D)
        else (s_A, s_B, s_C, s_D)

    out_A = s_A
    out_B = s_B
    out_C = s_C
    out_D = s_D
}

component hash_output() {
    input ready : Bool
    input a : UInt
    input b : UInt
    input c : UInt
    input d : UInt
    state saved_hash = start_hash : Hash
    state output_ctr = 0 : Unsigned 2
    output hash_word : Maybe UInt

    saved_hash' = if ready
        then ((a,b,c,d)) `add_hash` saved_hash
        else saved_hash

    should_emit = ready || output_ctr /= 0

    output_ctr' = if should_emit
        then output_ctr + 1
        else output_ctr

    hash_word = if should_emit
        then case output_ctr of
            0 -> Just (hash_a saved_hash')
            1 -> Just (hash_b saved_hash')
            2 -> Just (hash_c saved_hash')
            3 -> Just (hash_d saved_hash')
        else Nothing

}


component load_balance_4() {
    input msg_words : Maybe UInt
    state word_ctr = 0 : Unsigned 4
    output word_out_1 : Maybe UInt
    output word_out_2 : Maybe UInt
    output word_out_3 : Maybe UInt
    output word_out_4 : Maybe UInt

    word_out_1 = if turn == 0 then msg_words else Nothing
    word_out_2 = if turn == 1 then msg_words else Nothing
    word_out_3 = if turn == 2 then msg_words else Nothing
    word_out_4 = if turn == 3 then msg_words else Nothing

    word_ctr' = case msg_words of
```

```
        (Just _) -> word_ctr + 1
        Nothing -> word_ctr

    turn = word_ctr `shiftR` 2
}

component cat_4_maybes() {
    input hash_1 : Maybe UInt
    input hash_2 : Maybe UInt
    input hash_3 : Maybe UInt
    input hash_4 : Maybe UInt
    output hash : Maybe UInt

    hash = case hash_1 of
        (Just hash) -> Just hash
        Nothing -> case hash_2 of
            (Just hash) -> Just hash
            Nothing -> case hash_3 of
                (Just hash) -> Just hash
                Nothing -> case hash_4 of
                    (Just hash) -> Just hash
                    Nothing -> Nothing
}
```

md5_reuse.expi

```
system in (100, 80) at (2, 2) {
    input msg_words : Maybe UInt
    output hashes : Maybe UInt

    -- Compile this design with "--lpf-allow-unconstrained" in Nextpnr.hs
    -- There aren't really 33 pins for input and output available on the ECP5,
    -- So there should be some serdes components included as well, but that was
    -- not important for the research.

    msg_words->hash1.data
    hash1.hash->hashes

    balance is load_balance_4 in (100, 80) at (0, 0)
    -- in (3, hash1.h + hash2.h + hash3.h + hash4.h) at (hash1.x - 3, 0)

    msg_words->balance.msg_words
    balance.word_out_1->hash1.data
    balance.word_out_2->hash2.data
    balance.word_out_3->hash3.data
    balance.word_out_4->hash4.data

    cat is cat_4_maybes in (100, 80) at (0, 0)
    -- in (3, balance.h) at (hash1.x + hash1.w, 0)

    cat.hash->hashes
    hash1.hash->cat.hash_1
    hash2.hash->cat.hash_2
    hash3.hash->cat.hash_3
    hash4.hash->cat.hash_4

    hash1 in (21, 16) at (0, 0) {
        input data : Maybe UInt
        output hash : Maybe UInt

        calculator is calculator in (hash1.w, hash1.h) at (0, 0)
        const_store is const_store in (msg.w + 4, 10) at (8, 0)

        abcd is abcd_store in (hash1.w, hash1.h) at (0, 0)
        fabcd is fabcd_update in (14, hash1.h) at (outputter.w, 0)

        msg is message_store in (hash1.w - 12, hash1.h) at (12, 0)

        data->msg.data_in

        abcd.enable<-const_store.is_hashing
        abcd.out_A->fabcd.in_A
        abcd.out_B->fabcd.in_B
        abcd.out_C->fabcd.in_C
        abcd.out_D->fabcd.in_D
        const_store.s->fabcd.s
        const_store.k->fabcd.k
        const_store.stall<-msg.stall
        msg.m->fabcd.mg
        calculator.out_F->fabcd.f

        calculator.out_g->msg.g
        const_store.out_i->calculator.i

        abcd.out_A->calculator.a
        abcd.out_B->calculator.b
        abcd.out_C->calculator.c
        abcd.out_D->calculator.d

        abcd.in_A<-fabcd.out_A
        abcd.in_B<-fabcd.out_B
        abcd.in_C<-fabcd.out_C
        abcd.in_D<-fabcd.out_D

        outputter is hash_output in (5, hash1.h) at (0, 0)
        outputter.a<-fabcd.out_A
        outputter.b<-fabcd.out_B
        outputter.c<-fabcd.out_C
        outputter.d<-fabcd.out_D
        outputter.ready<-const_store.ready

        hash<-outputter.hash_word
    }

    -- square config
    -- hash2 is hash1 in (hash1.w, hash1.h) at (hash1.x, hash1.y + hash1.h)
    -- hash3 is hash1 in (hash1.w, hash1.h) at (hash1.x + hash1.w, hash1.y)
    -- hash4 is hash1 in (hash1.w, hash1.h) at (hash3.x, hash3.y + hash3.h)

    -- line config
    hash2 is hash1 in (hash1.w, hash1.h) at (hash1.x, hash1.y + hash1.h)
    hash3 is hash1 in (hash1.w, hash1.h) at (hash1.x, hash2.y + hash2.h)
    hash4 is hash1 in (hash1.w, hash1.h) at (hash1.x, hash3.y + hash3.h)
}
```

A.3 4×4 Manycore

manycore.expc

```
haskell {

(>>>) :: Bits a => a -> Int -> a
(>>>) = shiftR

(<<<) :: Bits a => a -> Int -> a
(<<<) = shiftL

type Int8 = Unsigned 8
type PC = Unsigned 5

type InstructionWord = Unsigned 8

type RegID = Unsigned 2
type Immediate = Unsigned 5

data Instruction
    = ReadFIFO RegID
    | WriteFIFO RegID
    | Add RegID RegID RegID
    | Move RegID RegID
    | Branch RegID RegID
    | LoadImm RegID Immediate
    deriving (Show, Generic, NFDataX, BitPack)

nop = Move 0 0

data FIFOCommand
    = FIFO_Write Int8
    | FIFO_Read
    | FIFO_Nothing
    deriving (Show, Generic, NFDataX, BitPack)

type ReadRegs = (RegID, RegID)
type WriteReg = Maybe (Int8, RegID)
```

```haskell
type RegisterFile = Vec 3 Int8

decode :: InstructionWord -> Instruction
decode word = if (bit 7)
    then LoadImm immreg imm
    else if not (bit 6)
        then Add reg45 reg23 reg01
        else if (bit 5)
            then if bit 4
                then WriteFIFO reg23
                else ReadFIFO reg23
            else if bit 4
                then Branch reg23 reg01
                else Move reg23 reg01
    where
        bit = testBit word
        -- the .&. is probably unnecessary due to the resize
        imm = resize $ word .&. 0b11111
        immreg = resize $ (word >>> 5) .&. 0b11
        reg01 = resize $ word .&. 0b11
        reg23 = resize $ (word >>> 2) .&. 0b11
        reg45 = resize $ (word >>> 4) .&. 0b11

encode :: Instruction -> InstructionWord
encode instr = case instr of
    ReadFIFO id ->
        0b0110_0000 .|. (resize id <<< 2)
    WriteFIFO id ->
        0b0111_0000 .|. (resize id <<< 2)
    Add left right dest ->
        0b0000_0000 .|. (resize left <<< 4) .|. (resize right <<< 2) .|. (resize dest)
    Move from to ->
        0b0100_0000 .|. (resize from <<< 2) .|. (resize to)
    Branch cond addr ->
        0b0101_0000 .|. (resize cond <<< 2) .|. (resize addr)
    LoadImm reg imm ->
        0b1000_0000 .|. (resize reg <<< 5) .|. (resize imm)

empty_regs :: RegisterFile
empty_regs = 0:>0:>0:>Nil

type Program = Vec 32 InstructionWord

default_prog :: Program
default_prog = map encode $ program ++ repeat (nop)
    where
        program =
            (LoadImm 1 1):>     -- 0 Load non-zero value in reg1 to avoid jumping
            (LoadImm 3 7):>     -- 1 Set jump addr to write sequence
            (ReadFIFO 1):>      -- 2 read from fifo
            (Branch 1 3):>      -- 3 if we receive zero, jump to write sequence
            (Add 1 2 2):>        -- 4 otherwise, add input to sum reg
            (LoadImm 3 1):>      -- 5 Set jump addr to start of program
            (Branch 0 3):>       -- 6 Always jump to address 1, i.e. read again

            (LoadImm 3 31):>     -- 7 Set destination to (4, 5)
            (LoadImm 1 31):>     -- 8 We need this convoluted way, since (4, 5) is 69
            (Add 1 3 3):>        -- 9 as an encoded value, which does not fit in 5 bits imm_values...
            (LoadImm 1 7):>      -- 10 31 + 31 + 7 = 69, probably better to use the 5th loadimm bit for
            (Add 1 3 3):>        -- 11 load upper vs load lower, but meh
            (WriteFIFO 3):>      -- 12 write destination
            (WriteFIFO 2):>      -- 13 write sum value
            (LoadImm 3 15):>     -- 14 set jump addr to end program
            (Branch 0 3):>       -- 15 End program, infinite loop
            Nil


writes :: FIFOCommand -> Maybe Int8
writes (FIFO_Write a) = Just a
writes _ = Nothing

type DataQueue = Vec 8 Int8
type DataPtr = Unsigned 3

empty_queue :: DataQueue
empty_queue = 1:>2:>3:>4:>5:>6:>7:>8:>Nil

type PacketQueue = Vec 4 Packet
data Packet = Packet Location Int8
    deriving (Show, Generic, NFDataX, BitPack)
type Location = (Unsigned 4, Unsigned 4)

empty_packet_queue :: PacketQueue
empty_packet_queue = zr:>zr:>zr:>zr:>Nil
    where
        zr = Packet (0, 0) 0

empty_packet :: Packet
empty_packet = Packet (0, 0) 0

empty_loc :: Location
empty_loc = (0, 0)

nothing = Nothing

data PacketControlState = Idle | GotLoc Location | Ready Location Int8
    deriving (Show, Generic, NFDataX, BitPack)

int2loc :: Int8 -> Location
int2loc v = (resize $ v >>> 4, resize v)

loc2int :: Location -> Int8
```

```
loc2int (x, y) = resize x <<< 4 .|. resize y

data SerializerState = SerIdle | SerSendLoc Packet | SerSendVal Packet
    deriving (Show, Generic, NFDataX, BitPack)

data Direction = North | East | South | West | Core
    deriving (Show, Generic, NFDataX, BitPack, Eq)

-- A test case for the full manycore.
mc_input :: [Maybe Int8]
mc_input = [
        Just 17, Just 5,
        Nothing, Nothing,
        Just 18, Just 13,
        Nothing, Nothing,
        Nothing, Nothing,
        Nothing, Nothing,
        Just 17, Just 5,
        Just 18, Just 13,
        Nothing, Nothing,
        Just 17, Just 0,
        Just 18, Just 0
    ] L.++ (L.take 1000 $ L.repeat Nothing)

}

component datapath() {
    input instr_word : InstructionWord
    input reg_a : Int8
    input reg_b : Int8
    input fifo_val : Maybe Int8

    state delay = 0 : InstructionWord
    state pc = 0 : PC

    output read_ids : ReadRegs
    output update_regs : WriteReg

    output fifo_cmd : FIFOCommand
    output pc_out : PC
    -- 5x4

    instruction = decode instr_word
    delay' = instr_word

    read_ids = case instruction of
        (ReadFIFO _) -> (0, 0)
        (WriteFIFO reg) -> (0, reg)
        (Add left right _) -> (left, right)
        (Move from _) -> (from, 0)
        (Branch cond addr) -> (cond, addr)
```

```
            (LoadImm _ _) -> (0, 0)

    delayed_instr = decode delay

    update_regs = case instruction of
        (ReadFIFO dest) -> case fifo_val of
            Just v -> Just (v, dest)
            Nothing -> Nothing
        (WriteFIFO _) -> Nothing
        (Add _ _ dest) -> Just (reg_a + reg_b, dest)
        (Move _ to) -> Just (reg_a, to)
        (Branch _ _) -> Nothing
        (LoadImm reg imm) -> Just (resize imm, reg)

    update_pc = case instruction of
        (Branch _ _) -> if reg_a == 0
            then Just $ resize reg_b
            else Nothing
        (ReadFIFO _) -> case fifo_val of
            Nothing -> Just pc


    fifo_cmd = case instruction of
        (ReadFIFO _) -> FIFO_Read
        (WriteFIFO _) -> FIFO_Write reg_b
        _ -> FIFO_Nothing


    pc' = case instruction of
        (Branch _ _) -> if reg_a == 0
            then resize reg_b
            else pc + 1
        (ReadFIFO _) -> case fifo_val of
            Nothing -> pc -- block on no value
            _ -> pc + 1
        _ -> pc + 1

    pc_out = pc

}

component registers() {
    input read_ids : ReadRegs
    input write : WriteReg

    state regs = empty_regs : RegisterFile

    output reg_a : Int8
    output reg_b : Int8

    -- 4x4

    regs' =
        case write of
            Just (value, regid) -> if regid == 0
                then regs
                else replace (regid - 1) value regs
            Nothing -> regs

    (reg_a_id, reg_b_id) = read_ids

    reg_a = if reg_a_id == 0
        then 0
        else regs !! (reg_a_id - 1)
    reg_b = if reg_b_id == 0
        then 0
        else regs !! (reg_b_id - 1)
}

component prog_mem() {
    input pc : PC
    state program = default_prog : Program
    output instr : InstructionWord
    -- voor default prog, moeilijk weinig: like 3x3, maar wsch in general een stuk meer nodig

    instr = program !! pc
    program' = program
}

component queue_controller() {
    input proc_cmd : FIFOCommand
    input route_cmd : FIFOCommand
    input router_read : Bool

    input outgoing_empty : Bool
    input outgoing_value : Packet
    output outgoing_read : Bool
    output read_packet : Maybe Packet

    input incoming_value : Int8
    input incoming_empty : Bool
    output incoming_read : Bool
    output incoming_write : Bool
    output incoming_datain : Int8
    output read_value : Maybe Int8

    -- 11 LUTs, ze lagen wat verspreid, niet veel iig.

    incoming_read = case proc_cmd of
        FIFO_Read -> True && (not incoming_empty)
        _ -> False

    incoming_write = case route_cmd of
        FIFO_Write _ -> True
        _ -> False

    incoming_datain = case route_cmd of
        FIFO_Write v -> v
        _ -> 0

    outgoing_read = router_read && (not outgoing_empty)

    read_value = if incoming_read
        then Just incoming_value
        else Nothing

    read_packet = if outgoing_read
        then Just outgoing_value
        else Nothing
}



component in_fifo() {
    input datain : Int8
    input write : Bool
    input read : Bool
    state rpntr = 0 : Unsigned 4
    state wpntr = 0 : Unsigned 4
    state elms = empty_queue : DataQueue
    output dataout : Int8
    output empty : Bool
    -- output full : Bool
    -- 6x5, at least

    -- https://github.com/clash-lang/clash-compiler/blob/master/examples/Fifo.hs
    wpntr' | write     = wpntr + 1
           | otherwise = wpntr
    rpntr' | read      = rpntr + 1
           | otherwise = rpntr

    mask  = resize (maxBound :: Unsigned 3)
    wind  = wpntr .&. mask
    rind  = rpntr .&. mask

    elms' | write     = replace wind datain elms
          | otherwise = elms

    n = 3

    empty = wpntr == rpntr
    full  = (testBit wpntr n) /= (testBit rpntr n) &&
            (wind == rind)

    dataout = elms !! rind

}

component packet_queue() {
    input datain : Maybe Packet
    input read : Bool
    state rpntr = 0 : Unsigned 3
    state wpntr = 0 : Unsigned 3
    state elms = empty_packet_queue : PacketQueue
    output dataout : Packet
    output empty : Bool
    -- output full : Bool
    -- 5x5

    write = case datain of
        Just _ -> True
        Nothing -> False

    -- https://github.com/clash-lang/clash-compiler/blob/master/examples/Fifo.hs
    wpntr' | write     = wpntr + 1
           | otherwise = wpntr
    rpntr' | read      = rpntr + 1
           | otherwise = rpntr

    mask  = resize (maxBound :: Unsigned 2)
    wind  = wpntr .&. mask
    rind  = rpntr .&. mask

    elms' | write     = case datain of
                          (Just v) -> replace wind v elms
           | otherwise = elms

    n = 3

    empty = wpntr == rpntr
    full  = (testBit wpntr n) /= (testBit rpntr n) &&
            (wind == rind)

    dataout = elms !! rind
}

component packet_control() {
    input proc_cmd : FIFOCommand
    state ctrl_state = Idle : PacketControlState
    output packet : Maybe Packet
    -- 3x3 is ruim

    ctrl_state' = case ctrl_state of
        Idle -> case proc_cmd of
```

```
            (FIFO_Write v) -> GotLoc (int2loc v)
            _ -> Idle
        GotLoc l -> case proc_cmd of
            (FIFO_Write v) -> Ready l v
            _ -> GotLoc l
        Ready _ _ -> Idle

    write = case ctrl_state of
        Ready _ _ -> True
        _ -> False

    packet = case ctrl_state of
        Ready l v -> Just $ Packet l v
        _ -> Nothing

}



component pkt_ser() {
    input empty_send_queue       : Bool
    input packet                 : Packet

    state ser_state = SerIdle    : SerializerState
    state send_loc = True        : Bool

    output value                 : Maybe Int8
    output read_from_send_queue  : Bool

    send_loc' = not send_loc

    ser_state' = case ser_state of
        SerIdle -> if read_from_send_queue
            then SerSendLoc packet
            else SerIdle
        SerSendLoc p -> SerSendVal p
        SerSendVal p -> if read_from_send_queue
            then SerSendLoc packet
            else SerIdle

    value = case ser_state' of
        SerIdle -> Nothing
        SerSendLoc (Packet loc _) -> Just $ loc2int loc
        SerSendVal (Packet _ val) -> Just val

    read_from_send_queue = send_loc && not empty_send_queue
}

component pkt_des() {
    input value : Maybe Int8
    state recv_loc = True : Bool
    state prev_value = Nothing : Maybe Int8
    output packet : Maybe Packet

    recv_loc' = not recv_loc
    prev_value' = value

    packet = if recv_loc
        then Nothing
        else case value of
            -- if this fromjust triggers, input data is out of sync with the des
            (Just value) -> case prev_value of
                (Just loc) -> Just $ Packet (int2loc loc) (value)
            Nothing -> Nothing

}

component direction_decider() {
    input my_x       : Unsigned 4
    input my_y       : Unsigned 4
    input from_north : Maybe Packet
    input from_east  : Maybe Packet
    input from_south : Maybe Packet
    input from_west  : Maybe Packet
    input from_core  : Maybe Packet

    state read_timer = 0 : Unsigned 2

    output to_north  : Maybe Packet
    output to_east   : Maybe Packet
    output to_south  : Maybe Packet
    output to_west   : Maybe Packet
    output to_proc   : FIFOCommand
    output ready     : Bool

    -- expects to be have my_x and my_y driven by a constant signal.

    to_north = pick_packet North
    to_east = pick_packet East
    to_south = pick_packet South
    to_west = pick_packet West
    to_proc = to_fifo $ pick_packet Core
    ready = read_timer == 0

    read_timer' = read_timer + 1

    dest (Just (Packet (x, y) _))
        | x > my_x = Just East
        | x < my_x = Just West
        | y > my_y = Just South
        | y < my_y = Just North
        | x == my_x && y == my_y = Just Core
    dest Nothing = Nothing

    -- silently drops packets :(
    pick_packet dir = if dest from_core == Just dir
        then from_core
        else if dest from_north == Just dir
            then from_north
            else if dest from_east == Just dir
                then from_east
                else if dest from_south == Just dir
                    then from_south
                    else if dest from_west == Just dir
                        then from_west
                        else Nothing

    to_fifo pkt = case pkt of
        Just (Packet _ v) -> FIFO_Write v
        Nothing -> FIFO_Nothing
}

component cap() {
    input out    : Maybe Int8
    output inp : Maybe Int8

    inp = Nothing
}
```

manycore.expi

```
manycore in (100, 92) at (4, 4) {
    input job_data : Maybe Int8
    output sums_out : Maybe Int8

    job_data->core_layout.topleft_in
    core_layout.botright_out->sums_out

    core_layout in (119, 88) at (0, 0) {
        input topleft_in : Maybe Int8
        output botright_out : Maybe Int8

        topleft_in->col_1[1].west_i
        col_4[4].south_o->botright_out

        io_cap is cap in (core_layout.w, core_layout.h) at (0, 0)

        io_cap.out<-col_1[1].west_o
        io_cap.inp->col_4[4].south_i

        -- Column 1
        chain col_1 at (0, 0) {
            component = pru in (router.w, router.h),
            amount = 4,
            layout = vertical,
            chain_in = north_i,
            chain_out = south_o
        }
        -- chain can only do one way, so we need to do the other manually
        col_1[4].north_o->col_1[3].south_i
        col_1[3].north_o->col_1[2].south_i
        col_1[2].north_o->col_1[1].south_i

        -- Cannot propagate constants through chains, so we'll do it this way :/
        col_1[1].my_y<-(1)
        col_1[2].my_y<-(2)
        col_1[3].my_y<-(3)
        col_1[4].my_y<-(4)

        col_1[1].my_x<-(1)
        col_1[2].my_x<-(1)
        col_1[3].my_x<-(1)
        col_1[4].my_x<-(1)

        -- Column 1<=>Column 2 link
        col_1:east_o->col_2:west_i
        col_1:east_i<-col_2:west_o

        -- Column 2
        chain col_2 at (col_1_1.w, 0) {
            component = pru in (col_1_1.w, col_1_1.h),
            amount = 4,
            layout = vertical,
            chain_in = north_i,
            chain_out = south_o
        }

        col_2[4].north_o->col_2[3].south_i
        col_2[3].north_o->col_2[2].south_i
        col_2[2].north_o->col_2[1].south_i

        col_2[1].my_y<-(1)
        col_2[2].my_y<-(2)
        col_2[3].my_y<-(3)
        col_2[4].my_y<-(4)

        col_2[1].my_x<-(2)
        col_2[2].my_x<-(2)
        col_2[3].my_x<-(2)
        col_2[4].my_x<-(2)


        -- Column 2<=>Column 3 link
        col_2:east_o->col_3:west_i
        col_2:east_i<-col_3:west_o
```

```

        -- Column 3
        chain col_3 at (col_2_1.x + col_2_1.w, 0) {
            component = pru in (col_1_1.w, col_1_1.h),
            amount = 4,
            layout = vertical,
            chain_in = north_i,
            chain_out = south_o
        }

        col_3[4].north_o->col_3[3].south_i
        col_3[3].north_o->col_3[2].south_i
        col_3[2].north_o->col_3[1].south_i

        col_3[1].my_y<-(1)
        col_3[2].my_y<-(2)
        col_3[3].my_y<-(3)
        col_3[4].my_y<-(4)

        col_3[1].my_x<-(3)
        col_3[2].my_x<-(3)
        col_3[3].my_x<-(3)
        col_3[4].my_x<-(3)


        -- Column 3<=>Column 4 link
        col_3:east_o->col_4:west_i
        col_3:east_i<-col_4:west_o

        -- Column 4
        chain col_4 at (col_3_1.x + col_3_1.w, 0) {
            component = pru in (col_1_1.w, col_1_1.h),
            amount = 4,
            layout = vertical,
            chain_in = north_i,
            chain_out = south_o
        }

        col_4[4].north_o->col_4[3].south_i
        col_4[3].north_o->col_4[2].south_i
        col_4[2].north_o->col_4[1].south_i

        col_4[1].my_y<-(1)
        col_4[2].my_y<-(2)
        col_4[3].my_y<-(3)
        col_4[4].my_y<-(4)

        col_4[1].my_x<-(4)
        col_4[2].my_x<-(4)
        col_4[3].my_x<-(4)
        col_4[4].my_x<-(4)
```

```


    -- Caps on unused sides of a PRU:
    repeat caps at (0, 0) {
        component = cap in (core_layout.w, core_layout.h),
        amount = 14,
        layout = identical
    }

    -- left side of manycore
    col_1[2-4]:west_i<-caps[1-3]:inp
    col_1[2-4]:west_o->caps[1-3]:out

    -- bottom
    col_1[4].south_i<-caps[4].inp
    col_1[4].south_o->caps[4].out
    col_2[4].south_i<-caps[5].inp
    col_2[4].south_o->caps[5].out
    col_3[4].south_i<-caps[6].inp
    col_3[4].south_o->caps[6].out

    -- top
    col_1[1].north_i<-caps[7].inp
    col_1[1].north_o->caps[7].out
    col_2[1].north_i<-caps[8].inp
    col_2[1].north_o->caps[8].out
    col_3[1].north_i<-caps[9].inp
    col_3[1].north_o->caps[9].out
    col_4[1].north_i<-caps[10].inp
    col_4[1].north_o->caps[10].out

    -- right
    col_4[1-4]:east_o->caps[11-14]:out
    col_4[1-4]:east_i<-caps[11-14]:inp


    -- The PRU design which is the basis of the manycore.
    unplaced pru in (28, 14) {
        input north_i : Maybe Int8
        input east_i : Maybe Int8
        input south_i : Maybe Int8
        input west_i : Maybe Int8
        output north_o : Maybe Int8
        output east_o : Maybe Int8
        output south_o : Maybe Int8
        output west_o : Maybe Int8

        input my_x : Unsigned 4
        input my_y : Unsigned 4

        router.north_i<-north_i
        router.east_i<-east_i
        router.south_i<-south_i
        router.west_i<-west_i

        router.north_o->north_o
        router.east_o->east_o
        router.south_o->south_o
        router.west_o->west_o

        router.my_x<-my_x
        router.my_y<-my_y

        router.route_cmd->core.route_cmd
        router.ready_for_packet->core.read_packet
        router.packet_from_core<-core.packet

        router in (30, 22) at (0, 0) {
            input north_i : Maybe Int8
            input east_i : Maybe Int8
            input south_i : Maybe Int8
            input west_i : Maybe Int8

            output north_o : Maybe Int8
            output east_o : Maybe Int8
            output south_o : Maybe Int8
            output west_o : Maybe Int8

            output route_cmd : FIFOCommand
            output ready_for_packet : Bool
            input packet_from_core : Maybe Packet

            input my_x : Unsigned 4
            input my_y : Unsigned 4

            unplaced send_queue in (router.w, router.h) {
                input packet : Maybe Packet
                output value : Maybe Int8

                queue is packet_queue in (router.w, router.h) at (0, 0)
                serializer is pkt_ser in (router.w, router.h) at (0, 0)

                queue.datain<-packet

                serializer.value->value
                serializer.packet<-queue.dataout
                serializer.empty_send_queue<-queue.empty
                serializer.read_from_send_queue->queue.read
            }

            repeat deserializers at (0, 0) {
                component = pkt_des in (router.w, router.h),
                amount = 4,
                layout = identical
            }

            deserializers[1].value<-north_i
            deserializers[1].packet->dir_dec.from_north
            deserializers[2].value<-east_i
            deserializers[2].packet->dir_dec.from_east
            deserializers[3].value<-south_i
            deserializers[3].packet->dir_dec.from_south
            deserializers[4].value<-west_i
            deserializers[4].packet->dir_dec.from_west

            dir_dec is direction_decider in (router.w, router.h) at (0, 0)
            dir_dec.ready->ready_for_packet
            dir_dec.to_proc->route_cmd
            dir_dec.my_x<-my_x
            dir_dec.my_y<-my_y
            dir_dec.from_core<-packet_from_core

            repeat queues at (0, 0) {
                component = send_queue in (router.w, router.h),
                amount = 4,
                layout = identical
            }

            queues[1].packet<-dir_dec.to_north
            queues[1].value->north_o
            queues[2].packet<-dir_dec.to_east
            queues[2].value->east_o
            queues[3].packet<-dir_dec.to_south
            queues[3].value->south_o
            queues[4].packet<-dir_dec.to_west
            queues[4].value->west_o

        }


        core in (router.w, router.h) at (0, 0) {
            input route_cmd : FIFOCommand
            input read_packet : Bool
            output packet : Maybe Packet

            processing.instr<-progmem.instr
            processing.pc->progmem.pc

            processing in (core.w, core.h) at (0, 0) {
                input instr : InstructionWord
                input fifo_val : Maybe Int8
                output pc : PC
                output fifo_cmd : FIFOCommand

                instr->datapath.instr_word
                datapath.fifo_cmd->fifo_cmd
                datapath.pc_out->pc

                regfile is registers in (core.w, core.h) at (0, 0)
                datapath is datapath in (core.w, core.h) at (0, 0)

                datapath.reg_a<-regfile.reg_a
                datapath.reg_b<-regfile.reg_b
                datapath.fifo_val<-fifo_val

                regfile.read_ids<-datapath.read_ids
                regfile.write<-datapath.update_regs
            }

            progmem is prog_mem in (core.w, core.h) at (0, 0)

            route_cmd->router_comm.route_cmd
            processing.fifo_cmd->router_comm.proc_cmd
            router_comm.packet->packet
            router_comm.read_packet<-read_packet
            processing.fifo_val<-router_comm.value

            router_comm in (core.w, core.h) at (0, 0) {
                input proc_cmd : FIFOCommand
                input route_cmd : FIFOCommand
                input read_packet : Bool
                output value : Maybe Int8
                output packet : Maybe Packet

                ctrl is queue_controller in (core.w, core.h) at (0, 0)
                ctrl.proc_cmd<-proc_cmd
                ctrl.route_cmd<-route_cmd
                ctrl.router_read<-read_packet

                in_queue is in_fifo in (core.w, core.h) at (0, 0)
                in_queue.datain<-ctrl.incoming_datain
                in_queue.write<-ctrl.incoming_write
                in_queue.read<-ctrl.incoming_read
                ctrl.incoming_value<-in_queue.dataout
                value<-ctrl.read_value
                ctrl.incoming_empty<-in_queue.empty

                out_queue is packet_queue in (core.w, core.h) at (0, 0)
                packet_control is packet_control in (out_queue.w, out_queue.h) at (out_queue.x, out_
                packet_control.proc_cmd<-proc_cmd
                out_queue.datain<-packet_control.packet
                out_queue.read<-ctrl.outgoing_read
                out_queue.empty->ctrl.outgoing_empty
                out_queue.dataout->ctrl.outgoing_value
```

```
                    packet<-ctrl.read_packet
                }

            }
        }

    }
}
```