Article

Structured Design of Complex Hardware Microarchitectures Based on Explicit Generic Implementations of Custom Microarchitectural Mechanisms

Alexander Antonov

Faculty of Software Engineering and Computer Systems, ITMO University, 197101 Saint-Petersburg, Russia; antonov@itmo.ru; Tel.: +7-911-700-83-61

Abstract: In typical computer system design flows, hardware microarchitecture is commonly considered a dynamically developing domain, where abstract mechanisms remain mostly in conceptual form and exhibit poor source code reuse in designs. However, many important basic mechanisms appear to be pervasive and persistent. This paper summarizes original efforts to propose generic implementations of microarchitectural mechanisms for in-order and out-of-order microarchitectures and various demo designs based on these implementations. The basic EDA components proposed for this design approach are “Kernel IP (KIP) cores”—open-source collection of programmable hardware generators inferred from microarchitectural templates. KIP cores feasibility for their role is demonstrated from the viewpoints of abstract representation of microarchitectural mechanisms, support of functional differentiation of designs, and generic optimizations. Experimental designing according to this approach has shown useful separation of concerns in hardware design activities, lower design effort, and increased formalization and code reuse.

Keywords: hardware generation; microarchitecture; RTL; KIP core; FPGA

1. Introduction

Modern systems-on-chip (SoCs) are designed as a mixture of non-programmable, slightly programmable, and fully programmable units (both general-purpose and specialized ones), utilizing multi-billion transistors budgets. Designing such devices relies on the stack of design languages, tools, and other technologies that are applied to design various subsystems on various levels of abstraction.

The efficient structuring of the design process is ensured by accurate separation of concerns, provided by responsible construction of standardized interfaces of various nature (languages, formats, software APIs, hardware communication protocols) and design artifacts compatible with these interfaces. These interfaces decouple well-proven generic design decisions and practices offered for broad reuse, and those implemented inside components, platforms, and tools. The former decisions are considered appropriate for gradual evolution, gentle customization, and sharing the findings across the industry, whereas the latter ones can exhibit rapid innovations and amenable to frequent, deep, and often secret modifications. In some cases, decisions from both groups can serve as a baseline for extension, alteration, and specialization.

Open standards and technologies, compared to proprietary ones, serve as “junction points” for common, widely known technical decisions and offer extra degrees of customizability that allows technical artifacts to be reused, assembled, and tweaked to greater extents to build real-life systems with various functionality and non-functional parameters. The following categories of technologies can be highlighted in this context.

(1) Processor instruction set architectures (ISA). Open examples include: OpenRISC, SPARC, and especially RISC-V. ISAs define an interface between programmable hardware
and software, decoupling CPU core designs and software infrastructure (including compilers, debuggers, middleware, etc.). This is considered as one of the most important interfaces in computer architecture [1]. Specifically, open RISC-V is often characterized as “Linux for hardware”—a free high-quality alternative to proprietary architectures.

(2) Hardware description languages (HDLs). HDLs provide general-purpose but low-level description of hardware, mostly in terms of individual combinational clouds, busses, registers, and memories (register transfer level, RTL). Limited numbers of HDLs are used commonly (SystemVerilog, VHDL, SystemC). However, (mostly) academic community proposes various “hardware generation” frameworks (Genesis2 [2], Chisel [3], SpinalHDL [4]) that allow to program construction of hardware and emit design specifications in standard HDLs. To simplify the development of custom tools, machine-oriented intermediate representations for hardware (mostly graph-based) have also been proposed. Examples are: FIRRTL [5], CIRCT [6], LLHD [7]. Designs in HDLs can be simulated using open-source simulators (Icarus, Verilator, SystemC kernel).

(3) High-level synthesis tools (HLS). These tools allow us to automate microarchitectural generation from behavioral specifications, usually written in subsets of common programming languages (e.g., C/C++). Open HLS tools (LegUp, Bambu, GAUT [8], Aetherling [9], Bluespec [10]) enable the customization of the synthesis process. Using standard compiler IRs from software domain (e.g., GCC GIMPLE, LLVM) facilitates customization even further. Although these flows have been expected to be an inevitable replacement for HDL-based ones for a long time, they still appear to be niche solutions.

(4) Open-source hardware implementations. These projects allow us to reuse and customize complex designs according to certain project needs. They include various types of blocks: processing (e.g., SonicBOOM [11], OpenXuantie [12], XiangShan [13] RISC-V CPUs), communication, memory, etc. Written in common HDLs (or as parametrizable generators), one problem is a steep learning curve when deep redefinition of some internal mechanisms is required (e.g., modified flow control, transaction scheduling, etc.). Since HDLs are of very low level relative to these aspects, the complexity of such modifications appears to be comparable with re-design.

(5) Open-source synthesis and implementation flows. These flows, though very restricted compared to proprietary ones, are still of great interest from the research viewpoint. Although contemporary HDLs do not adequately express the high-level ideas in hardware microarchitectures, commonly used open-source Yosys synthesis tool does not fully support even these HDLs, focusing mostly on legacy Verilog. However, compatible designs still can be even implemented as a real chip using existing open initiatives in ASIC design. Examples of these initiatives include SkyWater SKY130 PDK [14] and OpenROAD [15] projects.

Along with these numerous advances on all mentioned hardware-related levels, it can be noted that hardware microarchitecture level is typically not considered a standalone one in both open and proprietary flows and tools. As a result, it appears to be left for external, implicit, unformalized handling. Such handling of this level causes inadequately high design complexity, suboptimal code reuse, and demands for extra verification efforts.

In Section 2, the problem of insufficient handling of hardware microarchitecture is covered in more detail. Section 3 describes original “Kernel IP core” concept and open-source ActiveCore framework developed by the author addressing this problem. In Section 4, the exploration methodology of “Kernel IP cores” feasibility as basic components for “microarchitectural middleware” is presented. Section 5 summarizes experimental efforts of designing “Kernel IP cores” and designs based on them. Section 6 concludes the article.

2. Problem Statement

According to common computer architecture stack (see Figure 1), microarchitectural level resides between programmable processor ISA (effectively a format for executable code) and RTL defining the network of combinational elements, registers, memories, and other elements that collaborate to implement the functionality. Although the artifacts
belonging to “microarchitectural level” consistently appear on whiteboards, in natural language descriptions, and block diagrams in design documentation, this level still lacks its own, specialized, machine-readable formalisms, models, languages, and tools. When moving from architectural specs down to implementation, it is typically “leaped” directly from processor ISA (or other functional specs) directly to RTL. This “leap” is effectively considered the essence of the hardware design process.

![Diagram of computer architecture design levels](image)

**Figure 1.** “Microarchitectural” design level being implicit (“leaped”) in computer systems design flows.

Justification of “microarchitectural level” as an explicit, standalone level is not evident, since microarchitecture-related problems seem to be mostly covered by adjacent levels, both “higher” ones (processor ISAs, communication protocol specifications, accelerated HLS functions) and “lower” ones (RTL). The common issue of these higher levels is the pursuit to hide microarchitectural aspects from the designer, though some “knobs” still tend to emerge. Additionally, they appear to be very different in format and formalization levels. RTL processing, in turn, is fairly automated by design tools, but it treats design as a homogeneous set of structural modules with hardware interfaces and internal logic being disintegrated with any microarchitectural semantics. To evaluate microarchitecture-related coverage, we define several design aspects that are important for this level and assess how the developers handle them within common design flows.

1. **Taxonomy of microarchitectural components.** Complex hardware microarchitecture is a composition of controllers, datapath elements, storages, etc. that collaborate on given application problem. These elements might implement vastly different functions (control FSM, data computation, prefetching, synchronization with I/O). Processor ISAs generally abstract these details from the designer. Most high-level synthesis tools synthesize the design according to predefined template (e.g., FSMD: FSM with datapath iteratively computing given algorithm, usually in pipelined manner) and given set of tech library components. Additionally, custom functional unit support can be added (wrapped in custom functions calls). Some processor description languages such as CodAL (Codasip Studio) and nML allow us to capture selected information about the structure of processors. Common RTL HDLs do not provide means to bring any semantic differentiation for these different kinds of modules.

2. **Resource management.** In a general sense, these aspects are defined by control logic comprising the designs (e.g., hardware load balancers, arbitrers, etc.). On high levels, a certain amount of control might be exposed, but it is mostly imprecise. High-level synthesis systems perform automated allocation, scheduling, and binding, but also provide “pragma” options that allow us to “inject” some microarchitecture-related information (e.g., to unroll, merge, and pipeline loops). Additionally, the major limitation is that the schedule is static, which limits the feasibility of these tools for such designs as dynamically
scheduled CPUs [16]. There exist research works that attempt to embed the dynamism of data and control flows in synthesized designs [17]; however, it does not appear to be widely used in practice. In programmable processors, rules of functional unit activation, scheduling, and binding are typically hidden inside microarchitecture, though certain blocks (e.g., multipliers/divisors, FPUs, SIMD engines) have to be explicitly activated in software. Additionally, processor microarchitecture sometimes has to be considered for software optimization.

(3) Transaction flow control, synchronization, and routing. Any digital hardware design can be represented as a network of computational subblocks that exchange signals, data words, and messages. Transaction flows passing through this structure constitute computational process. The rules of subblocks integration and transaction management might be complex and multi-level and span the entire hardware structure (or the significant part of it). These rules include handshaking protocols, pipeline backpressure, buffering, policies of credits allocation and deallocation in credit-based flow control, shared resources arbitering, ordering points, points of no return for CPU instructions, etc. Networks-on-chip (NoCs) are “positive” examples in this context, since the primary focus in these designs is ensuring correct flow control, switching, and routing packets, and much attention is paid specifically to these issues. However, for other types of blocks, these rules are rarely formulated explicitly. Automated handling of these mechanisms can be implemented in certain projects using custom ad hoc tools (e.g., NoC generators [18]); however, no systemized, general-purpose methodology or tool for functionally differentiated designs appears to be proposed to date.

(4) Control structures configuration. To provide coherent circulation of transactions inside hardware microarchitecture, control logic has to be correctly configured and integrated. Examples are transaction ID width, credits number, buffer sizes, etc. Though this might seem a “local” problem, its importance is undeniable. These rules are implicit in RTL specifications and show very limited formalization, abstract implementation, and reuse. In turn, incorrect handling of this aspect might cause floating bugs that can be very difficult to identify, appear in long-term runs, but are destructive to system operation.

(5) Data synchronization. To correctly operate, actual copies of data have to be used by microarchitectural transactions. This includes RISC pipeline forwarding, register renaming (common for out-of-order microarchitectures), cache coherency in multicore, etc.

(6) Speculations. Elements of speculative execution exist in all categories of hardware blocks, including processors (branch prediction, return address stack, multipath execution, memory addresses prediction, register/memory value prediction), communicational (routing speculations), caches, stream buffers, etc. Complexity arises from speculation policies and recovery from misspeculations. Elements of speculation management exist in HLS (in terms of speculation “pragmas”) and CPUs (microarchitectural “hints”), however, being implicit in RTL implementations.

(7) Communication with external environment. External communication is typically defined in the form of common communication protocols (RAM, FIFO, AMBA AXI, Open Core Protocol). These protocols can be simple or very complicated: e.g., AMBA AXI specification contains hundreds of pages, including the coverage of split requests and responses, out-of-order transactions, coherency transactions, and atomic accesses. RTL design and verification of blocks and systems communicating through these channels can be error-prone and tedious, especially when communication mechanisms deeply integrate with internal computational process management. HLS tools provide certain pre-defined sets of interfaces for data exchange (however, they are typically not extendable with custom protocols).

Design decisions, algorithms, and practices related to microarchitecture have been in development for decades in parallel with ISA, HDLs, and manufacturing process advancements. Additionally, though processor microarchitectures evolve faster than architectures, generic microarchitectural mechanisms in their abstract formulations (e.g., concept of pipelining, backpressure mechanisms, out-of-order execution algorithms, etc.) appear
to be much more pervasive (implemented in various types of hardware) and persistent (remaining actual for decades). However, the lack of systematized handling methodology for these mechanisms motivates research efforts in their formalization, implementation in abstract form, synthesis and composition automation, and verification.

We refer EDA components and tools addressing these specific problems as “microarchitectural middleware” [19]. In further sections, original proposed format of EDA components for this problem is described and evaluated.

3. “Kernel IP Cores” Concept

To support the explicit implementation of custom execution models, inferred from microarchitectural templates, and hardware designs based on them, original open-source ActiveCore framework is being developed by the author [20]. ActiveCore is a software library of hardware generators providing custom programmable kernels constructed in accordance to certain microarchitectural template. Each such kernel (designated as “Kernel IP (KIP) core”) offers hardware description within custom execution model inferred from computational process organization inside hardware microarchitecture, applies microarchitecture-aware optimizations, and generates design specifications in standard RTL/HLS form [19,21,22].

KIP core’s execution model is intended to include:

- data types with specific order of initialization and assignment of values related to data circulation principles in hardware microarchitecture;
- auto-generated data structures and API for flow control, scheduling, communication, and synchronization “services” of the microarchitecture;
- microarchitecture-level event model and handler procedures selectively exposed for behavioral-style programming of custom application functions and mechanisms.

The distinctive feature of design languages provided by KIP cores in comparison to both basic HDLs and programming language in HLS tools is the adaptation of their execution model to generic principles of computational process organization, typical for certain class of microarchitectures. For example, whereas common HDLs handle individual signals according to discrete-event model of computation, the execution model of each KIP core exposes custom set of microarchitecture-level transactions, events, synchronization primitives, and other microarchitectural mechanisms. This technique provides precise control over the border between custom and automatically applied optimizations. In its turn, this enables both high-level specification capabilities and avoidance of “fighting with tools” in case their synthesis methods and hardware templates do not match with the expected hardware to be synthesized (e.g., static scheduling in HLS for dynamic CPU pipelines).

To streamline the implementation of custom data types, operations, and behavioral logic, hwast generic AST constructor has been designed. Custom data types, operations, and functions are implemented via inheritance from corresponding hwast base classes (see Figure 2).

KIP cores allocation for various hardware microarchitectures is considered useful for explicit decoupling of flow control, scheduling, communication, and synchronization mechanisms, typical for certain microarchitectures and functionally differentiated designs utilizing these mechanisms. In the current version of ActiveCore framework, the following KIP cores have been implemented (see Figure 3):

- Rtl—utility KIP core providing programmatic generation of behavioral-style RTL sources. Specifications can be exported to SystemVerilog HDL.
- Cyclix—utility KIP core providing programmatic generation of hardware specifications according to iterative, statically scheduled template. Specifications can be either exported to Rtl KIP (as synchronous circuits), or C++ code generated for Xilinx HLS.
- Pipex—KIP core providing programmatic generation of hardware with dynamic scalar in-order pipelined microarchitecture. Specifications can be exported to Cyclix KIP core.
- Reordex—KIP core providing programmatic generation of (co)processors with superscalar out-of-order (OoO) microarchitecture. Specifications can be exported to Cyclix KIP core.

**ActiveCore framework architecture**

![ActiveCore framework architecture](image)

**Figure 2.** ActiveCore framework architecture.

**Figure 3.** Overview of decoupled microarchitectural mechanisms and application logic in demo “Kernel IP cores”: Cyclix (a), Pipex (b), Reordex (c).

4. Evaluation Methodology of “Kernel IP Cores” as “Microarchitectural Middleware”

EDA Components

Allocation of separate design level for the developed automated generation assets can be justified by a set of the certain features. Here, these features are formulated.

1. Abstraction of microarchitectural mechanisms. Variability of custom types, operations, and processes allow “Kernel IP cores” to fill design capture with microarchitecture-related semantics. This allows us to specialize representation, optimization, and hardware generation methods according to this semantics, preventing the incorrect integration of logic pieces and automating the correct ones.

2. Functional differentiation. Most hardware specification and generation frameworks offer custom functionality specification within a certain, pre-defined computational model (e.g., LLVM in-memory IR for HLS tools [23], Chisel’s FIRRTL [5], LLHD, Amaranth’s FHDl AST [24], etc.), whereas the generators themselves are typically parametrized with data. The allocation of a separate design level requires building an interface that decouples specifications of custom functionality and implementation mechanisms. Hwast abstract behavioral AST in ActiveCore provides such generic, semantic-agnostic interface and stor-
(3) Generic configurations and optimizations. Separate design levels typically attempt to decouple variable functional specifications and generic configurations and optimizations that can be applied for functionally differentiated designs. Ideally, these optimizations have to be orthogonal to the functional part of design.

5. Results of Experimental Designing Based on “Kernel IP Cores”

Further, we dive deeper into the features of the designed demo KIP cores and, by these examples, show how the proposed design level features can be realized. Along with the framework, all these KIP cores and designs are available in open source [13].

5.1. Pipex

Pipex KIP core related to the early experiments within ActiveCore framework in decoupling of microarchitectural mechanisms and custom design utilizing these mechanisms. Pipex automates generation of hardware blocks with dynamic scalar in-order pipelined microarchitecture. Additionally, it provides constructs for data forwarding and variable-latency I/O synchronization. To a certain degree, this KIP core is inspired by TL-Verilog project [25] (though implemented differently). Previous iterations of Pipex generator have been described in previous works [19, 26]; however, we will cover its feasibility in terms of design level features proposed above.

Designing pipelines based on Pipex KIP core consists of defining staged and non-staged variables and functions that operate on these variables. These stages can be optionally buffered. The operation of design is driven by a stream of back-to-back transactions going through the pipeline. Control logic, including interfacing logic, transaction propagation, and backpressure in the case of stalls is generated automatically and abstracted in “application-level” logic of the pipeline.

Since functionality of stages is defined externally to Pipex KIP core, it can be used to implement functionally differentiated designs that share common pipelined organization. In Figure 4, three such examples are presented: sine wave generator based on Taylor series, Aquaris RISC-V CPU cores (with one to six pipeline stages and optional M and custom-0 pipelined coprocessor interfaces), and master/slave pipelines in Ariele full crossbar matrix generator. The CPU cores have been verified using open RISC-V compliance verification suite available at [https://github.com/riscv/riscv-compliance](https://github.com/riscv/riscv-compliance) (accessed on 12 February 2022). These generators allowed to hit (depending on configuration) up to 185 MHz on 7-series Xilinx FPGA device and above 90% per-core reuse of codebase between the CPUs and 63% between CPU and bus matrix generators [19].

Figure 4. Demo pipelined designs based on Pipex KIP core: sine wave generator based on Taylor series (a), RISC-V CPU core (b), and master and slave pipelines in full cross-bar (c).

For design optimization, there are a number of options that allow to tweak the design to different PPA ratios.
The first (automatic) micro-optimization is analyzing read and written staged variables and removing those that do not need to be preserved in certain stages.

The second (semi-manual) one is the redistribution of custom logic across pipeline stages. Figure 5 shows example of various staging for sine wave generator based on Taylor series. As shown in the Figure, different numbers of pipeline stages can be allocated, and the same application logic can be rebinded to these different stages. Staging and flow control logic is generated automatically by the Pipex generator. This enables fast design space exploration with minimal design effort. RISC-V CPU and bus matrix generators utilize the same capabilities plus automatic pipelined I/O transaction synchronization (synchronization of asynchronous pipelined I/O transactions with the main pipeline).

![Figure 5](image)

**Figure 5.** Example of staging options in sine wave generator based on Pipex KIP core: 2-stage (a), 4-stage (b), and 6-stage (c).

The final optimization “knob” available in Pipex is a selection between stall-based and credit-based flow control protocols. Figure 6 shows these two configuration options and simulation waveforms produced by the resulting designs. In stall-based design, when the response interface in the end is not ready, all stages are successively filled with transactions. In a credit-based design, the last buffered stage defines the size of automatically generated credit counter in the beginning of the pipeline. Only the number of transactions that can fill the last stage are allowed to pass the beginning stage of the pipeline. This scheme, though requiring additional logic, prevents stall signal accumulation hampering maximum frequency and resulting performance.
Figure 6. Flow control options provided in Pipex KIP core: stall-based (a), credit-based (b).

Additionally, Pipex implements a sanity check during the generation stage, preventing the subsequent stages directly reading the values in the previous stages, which would cause combinational loops.

To summarize the features of Pipex KIP core justifying it as a design level, Table 1 is shown.

Table 1. Summary of “microarchitectural middleware” features of Pipex KIP core.

<table>
<thead>
<tr>
<th>“Microarchitectural Middleware” Feature</th>
<th>Implementation in Pipex KIP Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction of microarchitectural mechanisms</td>
<td>Transaction flow control in dynamic scalar pipeline, I/O synchronization with pipeline</td>
</tr>
<tr>
<td>Functional differentiation</td>
<td>Fixed-function accelerators, CPU cores, communicational blocks</td>
</tr>
<tr>
<td>Generic configurations and optimizations</td>
<td>Staging, buffering, redundant state elimination, switchable flow control</td>
</tr>
</tbody>
</table>

To evaluate the improvement of project structure when using the proposed approach, code reuse characteristics for functionally differentiated designs are further evaluated. Statistic plugin for IntelliJ IDEA was used to count the lines of code (LOC). The results for Pipex for various designs based on it are summarized in Table 2.

Table 2. Code reuse evaluation for Pipex and designs based on it.

<table>
<thead>
<tr>
<th>Design Component</th>
<th>LOC</th>
<th>Pipex Code Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipex</td>
<td>1345</td>
<td>N/A</td>
</tr>
<tr>
<td>Taylor generator</td>
<td>73</td>
<td>95%</td>
</tr>
<tr>
<td>RISC-V CPU core generator</td>
<td>1271</td>
<td>51%</td>
</tr>
<tr>
<td>Cross-bar generator</td>
<td>321</td>
<td>81%</td>
</tr>
</tbody>
</table>

The presented results confirm drastic reduction in code size and significant (over 50%) code reuse for vastly different designs, which we consider unachievable for common HDL-based approaches.

5.2. Reordex

Reordex is a recently designed KIP core that addresses a generation of processors and coprocessors with superscalar out-of-order execution of transactions. The main intention behind this generator is to decouple transaction flow control, execution unit functionality, register management, and I/O communication.

The functionality of the design is specified as a set of untimed procedural blocks corresponding to various execution unit pipelines and communicating with the rest of design...
via latency-insensitive request–response interface. These procedures can be selectively exported to synthesizable SystemVerilog HDL and C++ sources for Xilinx HLS in arbitrary combinations. Flow control logic and integration logic is generated and integrated with the rest of the design automatically. Shared register states are specified as a special configuration object, available for direct reference in procedural blocks. Special configuration options define:

- Register file width;
- Architectural register file depth;
- Width of main pipeline datapath (fetch, decode, dispatch, reorder, commit);
- Register management policy (switchable between scoreboard and register renaming);
- All stage buffers sizes;
- RISC/coprocessor mode;
- Source data and register write transmission channels;
- Additional transaction flow elements passed from decoding stage to execution units.

In RISC processor mode, an additional decoding function that maps instruction code to Reordex resources should be provided.

Based on Reordex, two demo designs have been proposed: Citadel FPU coprocessor and Agenda RISC-V CPU core (see Figure 7). FPU coprocessor consists of five pipelines (two adding/subtraction, multiplication, division, FMA), whereas Agenda CPU has been programmed to support RISC-V (RV32IM) ISA. Both designs have been synthesized for 7-series Xilinx FPGA devices. Depending on the configuration, FPU hit up to 122 MHz on the Artix-7 (xc7a100t) device [14], and Agenda CPU hit up to 90 MHz on the Xilinx Kintex-7 (xc7k325t) device, which are typical numbers for complex out-of-order cores without FPGA-specific optimizations.

The first optimization provided in Reordex KIP core is automated issue logic generation. Depending on usage of operands by execution units, different operation firing logic has to be generated. Reordex automates the generation of this logic, not requiring any additional user intervention. The second optimization "knob" (exposed to the designer) is the configuration of the datapath width (frontend and reorder buffer). The final optimization option (exposed to the designer) is selection between two modes of register management: scoreboard (no register renaming) and register renaming. The last two optimization options are highlighted in Figure 8.

Summarizing the abovementioned information, Reordex provides the orthogonalization of selected microarchitectural mechanisms, including execution units functionality, shared resources, transaction flow control, datapath width, and register management policy, available for reconfiguration in arbitrary combinations. To summarize the features of Reordex KIP core justifying it as a design level, Table 3 is shown.
The proposed approach has shown differentiation and deep reconfigurability of designs with programmable hardware generators inferred from microarchitectural templates.

Table 3. “Microarchitectural middleware” design level features of Reordex KIP core.

<table>
<thead>
<tr>
<th>“Microarchitectural Middleware” Feature</th>
<th>Implementation in Reordex KIP Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction of microarchitectural mechanisms</td>
<td>Transaction flow control, transaction acquisition control, transaction issue control, register management</td>
</tr>
<tr>
<td>Functional differentiation</td>
<td>Coprocessors and programmable processors</td>
</tr>
<tr>
<td>Generic configurations and optimizations</td>
<td>Dynamic datapath width, switchable register management policy, other options</td>
</tr>
</tbody>
</table>

Similar to Pipex, evaluation of code reuse for Reordex and designs based on it are summarized in Table 4.

Table 4. Code reuse evaluation for Reordex and designs based on it.

<table>
<thead>
<tr>
<th>Design Component</th>
<th>LOC</th>
<th>Reordex Code Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reordex</td>
<td>2289</td>
<td>N/A</td>
</tr>
<tr>
<td>FPU coprocessor generator</td>
<td>48</td>
<td>98%</td>
</tr>
<tr>
<td>RISC-V OoO CPU core generator</td>
<td>1069</td>
<td>68%</td>
</tr>
</tbody>
</table>

The results of experimental designing based on Reordex KIP core also confirm the reduction in code size and increased code reuse, in this case, for differentiated designs with superscalar, out-of-order microarchitecture.

6. Discussion

Increasing demand for customized hardware platforms actualizes the development of design methods and tools that allow to fine-tune the hardware for specific applications and workload characteristics. Custom hardware designs, especially programmable ones, impose a significant complexity problem that is not adequately solved using mainstream design approaches and techniques.

To address this problem, the explicit allocation of the microarchitectural design level embodying well-known microarchitectural decisions is explored. The justification of this design level includes: (i) formulation of multiple aspects of hardware microarchitectures addressed by pervasive and persistent mechanisms that remain mostly in conceptual form and exhibit poor source code reuse in designs; (ii) experimental implementation and evaluation of abstract generic mechanisms for in-order and out-of-order microarchitectures. These implementations take the form of “Kernel IP (KIP) cores”—open-source collection of programmable hardware generators inferred from microarchitectural templates.

The experimental design of processing and communication IP blocks based on the proposed approach has shown differentiation and deep reconfigurability of designs with complex custom microarchitecture without quality of results compromises.

The proposed approach and implementation also have limitations and open issues that are planned to be addressed in future research. Arguably, the key one is directly caused by the idea of deep adaptation of design infrastructure to certain classes of microarchitectures and the mechanisms comprising them. This is a double-edged sword: multiple KIP cores offering specialized design languages and optimization techniques can lead to the
fragmentation of design optimizations and insufficient portability of design specifications. This actualizes a balanced approach to KIP cores specialization. The promising direction to address this issue might be accumulation and formalization of methods relevant to multiple KIP cores. Another approach might be combining multiple separate KIP cores in generalized ones reflecting common principles of their organization. This idea is similar to “profiles” in RISC-V ISA that reasonably combine selected RISC-V ISA extensions to reduce fragmentation.

Further improvements for the project include timing and other microarchitectural optimizations for hardware generated by existing “Kernel IP cores”. Additionally, exposing multiple optimization options actualized the development of design space exploration methods for these deeply configurable generators (e.g., ML-based approaches). In terms of verification, two points can be highlighted. Firstly, flexible hardware generators, in addition to deeper design adaptation, add extra dimensions of fuzziness during design space exploration, which helps to reveal more errors. Another direction is explicit decoupling of verification activities for “Kernel IP cores”, and designs based on them. This is verification-related research that is planned to be addressed in the nearest future.

**Funding:** This work was financially supported by Russian Science Foundation, grant No. 20-79-00219.

**Data Availability Statement:** The data presented in this study can be reproduced using the materials from public repository at [https://github.com/AntonovAlexander/activecore](https://github.com/AntonovAlexander/activecore) (accessed on 12 February 2022).

**Conflicts of Interest:** The author declares no conflict of interest.

**References**