A Configurable TLB Hierarchy for the RISC-V Architecture

Nikolaos Charalampos Papadopoulos, Vasileios Karakostas, Konstantinos Nikas,
Nectarios Koziris, Dionisios N. Pnevmatikatos
School of ECE, National Technical University of Athens
{ncpapad, vkarakos, knikas, nkozyris, pnevmati}@cslab.ece.ntua.gr

***Abstract*—The Rocket Chip Generator uses a collection of parameterized processor components to produce RISC-V-based SoCs. It is a powerful tool that can produce a wide variety of processor designs ranging from tiny embedded processors to complex multi-core systems. In this paper we extend the features of the Memory Management Unit of the Rocket Chip Generator and specifically the TLB Hierarchy. TLBs are essential in terms of performance because they mitigate the overhead of frequent Page Table Walks, but may harm the critical path of the processor due to their size and/or associativity. In the original Rocket Chip implementation the L1 Data/Instruction TLB is fully-associative and the shared L2 TLB is direct-mapped. We lift these restrictions and design and implement configurable, set-associative L1 and L2 TLB templates that can create any organization from direct-mapped to fully-associative to achieve the desired ratio of performance and resource utilization, especially for larger TLBs. We present the area for different configurations and evaluate the overall performance of our design using the SPEC2006 benchmark suite on the Xilinx ZCU102 FPGA. Our design is intended both for ASIC implementation and for FPGA-friendly soft processors. As FPGAs continue to increase in size, it becomes increasingly attainable and desirable to use configurable high-performance soft processors that can run full-fledged operating systems, especially for applications with large memory footprints.**

***Index Terms*—RISC-V, Rocket Chip Generator, Memory Management Unit, TLB Hierarchy, FPGA**

I. Introduction

FPGA designs often incorporate a number of general purpose soft processors. As the range of FPGA applications broadens and evolves, the performance demand from soft-processors will likely increase [1]–[7]. In addition, modern FPGA fabrics are growing in size thanks to technology improvements, and become capable of accommodating more complex and larger soft-processor designs. Hence, as FPGAs continue to increase in size, it becomes increasingly important to use configurable high-performance soft processors that can run full-fledged operating systems and that are tailored to the needs of the target applications.

The Rocket Chip Generator (RCG) [8] uses a collection of parameterizable processor components to produce RISC-V-based SoCs. While the RCG was initially intended for ASIC implementation, its adaptation by the community and academia has lead to prototypes in multiple FPGA platforms [9]–[11]. The RCG supports fully-fledged Unix-like operating systems, RISC-V extensions, and accelerators.

The RCG targets a wide range of application domains, ranging from embedded up to high-end systems. To support these diverse application domains, most components have been implemented as configurable templates in the Chisel [12] high-level hardware description language (HCL). However, some of the Rocket Chip Generator components are still missing support for configurability. In this paper we particularly focus on the Memory Management Unit (MMU) and specifically on the Translation Lookaside Buffer (TLB) hierarchy that lack such configurability support. TLBs are essential in terms of performance because they mitigate the overhead of frequent page table walks, but may harm the critical path of the processor due to their size and/or associativity.

In the original Rocket Chip implementation, only the number of TLB entries is configurable; the L1 Instruction and Data TLBs can only be fully-associative and the shared L2 TLB direct-mapped. However, that approach is not optimal for applications with large memory footprints that require larger TLB reach with many entries because: (i) increasing the number of the fully-associative L1 TLB significantly increases the critical path of the processor and can impact the operating frequency of the entire design, and (ii) a direct-mapped L2 TLB can experience many conflict misses, leaving significant room for application performance improvement with the use of increased associativity. Furthermore, for FPGA implementations high-associativity content-addressable memories (CAMs) may lead to poor resource usage and lower frequency. Lowering the associativity while keeping a large number of entries alleviates this issue without sacrificing performance. Hence, the lack of configurability in the TLB may limit the efficient applicability of RCG soft processors for applications with large memory footprints that stress the TLB hierarchy.

In this paper we lift these restrictions and design and implement configurable, set-associative L1 and L2 TLB templates that can create any organization from direct-mapped to fully-associative to achieve the desired ratio of performance and resource utilization, especially for larger TLBs. We also modify existing replacement policies to be compatible with our design, offering flexibility for performance and resource usage trade-offs. We use different L1/L2 TLB configuration scenarios to evaluate our design with benchmarks from SPEC2006int [13]. We show that performance improves by up to 15.4% for the largest evaluated TLB configuration, with minimal impact in resource usage.

978-1-7281-9902-3/20/$31.00 ©2020 IEEE
DOI 10.1109/FPL50879.2020.00024

In summary the main contributions of this paper are:

- We implement a fully configurable Instruction/Data L1 TLB and shared L2 TLB that can output any design from direct-mapped to fully-associative, lifting the initial restrictions of configurability only by the number of entries. This leads to better scaling of performance and resources, especially for large TLBs. We make our design publicly available[1] to enable further research on the active topic of virtual memory support for RISC-V.
- We present a case study in which we evaluate the performance and resource usage of the Rocket Chip [8] processor with different TLB configurations, by running benchmarks from SPEC2006int on the Xilinx ZCU102 FPGA.

II. Background

In this section we provide background information on Virtual Memory and the Rocket Chip Generator.

A. Virtual Memory

Virtual memory is an essential concept for processor design because it provides the illusion of a very large and private address space to each process running in the system. Virtual memory offers security through process isolation and also benefits programmer productivity since the operating system manages the memory mappings and the hardware accelerates the translations from the virtual to the physical address space.

RISC-V supports different Virtual Memory systems depending on the size of the address space (e.g., RV32 Sv32, RV64 Sv39/Sv48 [14]). In this paper we focus on RV64 Sv39 (39-bit address space) which supports 4KB base pages but also 2MB and 1GB super pages. The page table, that stores the memory mappings of each process, is implemented as a multi-level radix tree (3-level page table in RV64 Sv39). A processor register called `SATP` (Supervisor Address Translation and Protection register) holds the root of the page table. The physical address is obtained after performing a sequential lookup in each page table level. The page table walker (PTW) that performs the virtual-to-physical address translations is typically implemented in hardware for improved performance.

To accelerate address translation without accessing the page table on every memory reference, a Translation Lookaside Buffer (TLB) is used which keeps the recently used translations. The TLB lies on the critical path of the processor and as a result its latency and hit rate are essential for the overall performance. To overcome this problem without sacrificing the hit rate, multi-level TLB organizations are used; the first level TLB (L1) is usually small (32-128 entries) but very fast, while the second level TLB (L2) is usually larger (128-1024 entries) but slower. Finally, a Page Table Walk cache is usually implemented to hold non-leaf intermediate translations of the page table to avoid searching levels of the page table (TLBs hold the leaf translations). Figure 1 shows these structures.

[1]https://github.com/ncppd/rocket-chip

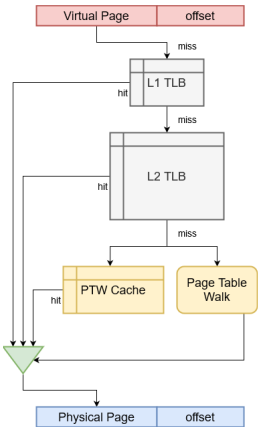

Fig. 1. Overview of the Memory Management Unit in Rocket Chip Generator.

B. Rocket Chip Generator

The Rocket Chip Generator (RCG) [8] generates RISC-V ISA [14], [15] based systems using Chisel. The RCG can also be considered as a library of processor parts that can easily be reused with any design written in Chisel. By default, the Rocket Chip Generator instantiates Rocket, an in-order core implementation, but also supports various core implementations including the BOOM out-of-order processor [16]. Rocket is a simple, 5-stage, in-order processor that implements the RISC-V ISA, including an MMU that supports page-based virtual memory, TLBs, instruction and data caches, and a frontend that features dynamic branch prediction with configurable sizes.

III. TLB Hierarchy Design

In this section we provide an overview of the original implementation of the Instruction/Data L1 and shared L2 TLB in the Rocket Chip Generator. Then, we present the design and implementation of our proposed configurable L1 and L2 TLB. Our design can output any organization ranging from direct-mapped up to fully-associative TLBs.

A. Original TLB overview

The L1 Instruction/Data TLBs are built based on the same Chisel template in the RCG and only have minor differences regarding access privileges to pages. The L2 TLB is shared among the L1 Instruction/Data TLBs and can contain both Instruction and Data page translations.

1) L1 TLB: The L1 Instruction/Data TLB stores the page translations in registers using Chisel's `Reg` element which can be synthesized to FPGA Flip-Flops; `Reg` creates a positive-edge-triggered register that outputs a copy of the input signal delayed by one clock cycle, depending on its activation signal. The original L1 TLB is fully-associative with configurable number of entries and a Pseudo-LRU replacement policy. The L1 TLB responds with a hit/miss indication on the next cycle.

2) L2 TLB: The shared L2 TLB stores the page translations using Chisel's `SeqMem`[2] construct, which can be

[2]`SeqMem` is renamed to `SyncReadMem` in Chisel3

synthesized to FPGA Block RAM. `SeqMem` basically creates a synchronous-read, synchronous-write memory, in this case with one read and one write port. The L2 TLB is direct-mapped with configurable number of entries. Because of the direct-mapped organization there is no need for a replacement policy. The L2 TLB informs of a hit/miss after one cycle because of the `SeqMem` construct. The requested value of a register is obtained during the same cycle. On the other hand, a request to the `SeqMem` will be delivered on the next cycle: `SeqMem` outputs to a register in order to perform a synchronous read operation. The L2 TLB keeps a separate *Valid-bit array* that indicates the valid entries; it is stored in `Reg` banks to avoid the cycle delay of the `SeqMem` construct.

Note that the Page Table Walk Chisel template incorporates the shared L2 TLB. The PTW is connected with the L1 Instruction and Data TLBs though a round-robin arbiter that selects the target virtual address to be translated.

3) Page Table Walk Cache: The PTW Cache is a small fully-associative cache that stores the non-leaf virtual-to-physical page translations. In this paper we focus on the TLBs and leave the PTW Cache for future work.

B. Configurable L1 TLB Architecture

Our design and implementation adheres to the requirements of the original implementation, i.e., uses (i) Chisel's `Reg` element for fast lookup time, and (ii) same Chisel template for both Instruction and Data TLBs respecting access privileges.

1) TLB Lookup: Whenever an address translation is requested, we obtain a *tag* and an *index* by splitting the *VPN*. Using the *index* we locate the target set and perform a fully-associative search that matches the *tag*.

2) TLB Refill: When a TLB refill is requested, we locate the target set that the VPN/PPN must be inserted using the *index*. We either select the first free open slot or perform a Pseudo-LRU replacement.

3) Replacement Policies: We modify the existing Pseudo-LRU replacement policy and implement a set-associative alternative. The random replacement policy was already supported. A random replacement policy is an attractive alternative option thanks to its simplicity; however, it may increase the TLB miss rate and hence degrade performance.

4) Flushing a L1 TLB entry: When the OS modifies the page table, the stale TLB entries must be flushed. This happens when the OS executes the `sfence.vma` to invalidate an entry. The flushing of the corresponding TLB entry is done by zeroing the valid bit of the specified entry.

C. Configurable L2 TLB Architecture

We now describe how lookups, refills, replacements, and flushes are handled in our configurable L2 TLB.

1) TLB lookup: The TLB lookup mechanism is similar to that of the L1 TLBs. The only difference is the lookup in the L2 TLB introduces an additional delay cycle because of the `SeqMem` construct. As a result we use registers to hold intermediate state.

2) TLB Refill: When a TLB refill is ordered, the L2 TLB handles it similarly with the L1 TLB. The only difference is the use of *masks* to update a specific way in a set. Masks are a feature of the `SeqMem` construct to ease updating specific indexes inside a set.

3) Replacement Policies: Both PseudoLRU and random replacement policies are implemented and can be used for the L2 TLB. In Section V we choose to evaluate our set-associative design using Random Replacement Policy in favor of area constraints.

4) Flushing a L2 TLB Entry: Flushing a TLB entry on a set-associative organization means that the entry must be located inside the selected set. In order to fetch the tags of the selected set, there must be a cycle delay because of the `SeqMem` construct. To overcome this overhead and keep the flushing mechanism simple, we select to flush the whole set.

IV. Methodology

In this section we describe our evaluation methodology, including the tools, metrics, and configurations.

A. Software and Hardware tools

We use the Rocket Chip Generator to instantiate a Rocket 5-stage in-order core on the Xilinx ZCU102 FPGA board. The board has an XCZU9EG FPGA with 548,160 configurable logic block (CLB) LUTs, 274,080 CLB FFs and 912 BRAMs [17]. We use Sifive's Freedom-U-SDK [18] which sets up a minimal Linux environment. The Rocket Chip SoC boots the lightweight Buildroot distribution on top of Linux kernel 4.15.0 with 4KB pages.

We use Vivado 2018.1 Design Suite to compile the bitstream and to get resource usage results. To evaluate the different TLB hierarchy scenarios, we run benchmarks from the SPEC2006int [13] with the test input set, due to the limited physical memory (512MB) that our Xilinx ZCU102 platform exposes to the PL. Finally, we use various L1 Instruction/Data TLB and shared L2 TLB configurations. In all configurations we use a 4-way 32KB instruction cache and a 4-way 16KB data cache.

B. Metrics

To evaluate our configurable TLB hierarchy we use the following metrics: (i) FPGA resource usage, i.e., flip-flops, look-up-tables (LUTs), and block RAM, (ii) TLB performance, i.e., TLB Misses-Per-Kilo-Instructions (MPKI), and (iii) system performance, i.e., Instructions-Per-Cycle (IPC), a performance metric that isolates the impact of TLB implementation on the critical path ignoring the processor frequency.

C. Configuration scenarios

We evaluate our modified set-associative design using different configurations for the L1 Instruction/Data TLB and shared L2 TLB. Table I summarizes the evaluated configurations. We choose these configurations to cover a range of systems from small and embedded up to modern high-performance general-purpose systems. The TLB reach (i.e.,

TABLE I
ROCKET CHIP L1 INSTRUCTION/DATA TLB AND SHARED L2 TLB CONFIGURATION SCENARIOS (ASSOCIATIVITY /SIZE)

Conf. No	DTLB	ITLB	L2 TLB	DTLB Reach	ITLB Reach	L2 TLB Reach
I	fully-associative, 32 entries	fully-associative, 32 entries	-	128KB	128KB	-
II	fully-associative, 32 entries	fully-associative, 32 entries	4-way, 128 entries	128KB	128KB	512KB
III	fully-associative, 32 entries	fully-associative, 32 entries	4-way, 512 entries	128KB	128KB	2MB
IV	8-way, 64 entries	8-way, 128 entries	8-way, 1024 entries	256KB	512KB	4MB
V	8-way, 128 entries	8-way, 64 entries	8-way, 1024 entries	512KB	256KB	4MB

number of entries × page size) covered by the L1 ranges from 128KB to 512KB, and for the L2 is up to 4MB. In the most lightweight configuration we choose not to include an L2 TLB to quantify the performance and area differences of the different configurations. Finally, in the most performant TLB configurations (Configurations IV, V) we swap the size of the Data and Instruction TLB to identify possible changes in performance without changing the L2 TLB.

To summarize, the configuration scenarios are chosen to resemble well-known architectures:

I. Vanilla Rocket Chip without L2 TLB
II. Vanilla Rocket Chip including small L2 TLB
III. ARM Cortex A57 [19]
IV. Intel Skylake [20]
V. Intel Skylake with swapped Instruction/Data TLB sizes.

V. RESULTS

A. Area and Frequency Results

Figure 2 shows the area results for the various configurations. We present the total area of the Rocket Chip SoC as reported by the Vivado 2018.1 Implementation stage. Note that the Instruction/Data L1 TLB structures use FFs and the shared L2 TLB uses BRAMs.

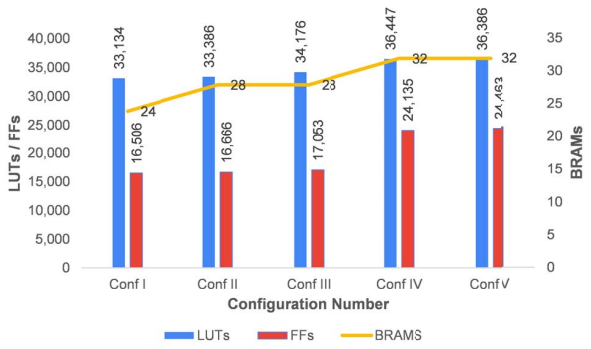

Fig. 2. Area details for different TLB configurations.

In the first three scenarios (Conf I, II, III) Vivado 2018.1 reports that the full Rocket Chip SoC occupies 12% of the total LUTs, 3% of the total FFs, and 3% of the total BRAMs of the Xilinx ZCU102. Moving from Configuration II to III there is a minor increase only in LUTs/FFs, while BRAM usage stays the same even if the L2 TLB is 4× larger than Configuration II. To achieve the best possible performance, Vivado uses 4 BRAMs, one for each of the 4 ways of the L2 TLB. Until the 4-BRAM size is exhausted, the resource usage is slightly increased, mainly for addressing purposes. Tuning up to the most performant configurations (Configuration IV, V) in terms of TLB hit rate, the Rocket Chip SoC occupancy increases to 13% for total LUTs, and 13% for total BRAM/FF usage.

TABLE II
MAXIMUM OPERATING FREQUENCY PER CONFIGURATION

Configuration	I	II	III	IV	V
Frequency (Mhz)	189	187	186	188	186

Table II shows the maximum frequency achieved with all configurations. The results show that the increase of the TLB resources with our design has low impact on the maximum operating frequency, ranging from 0.53% to 1.59%. In particular, Configuration IV has a 2× larger DTLB, 4× larger ITLB, and a 1024 entry L2 TLB, but exhibits only a 0.53% drop in frequency compared to Configuration I.

B. Performance Results

We now present the results of SPEC2006int benchmark suite that we obtained on the Xilinx ZCU102 FPGA Board. Figure 3 shows the results of aggregated MPKI in the L1 Instruction/Data TLBs for the various configurations. We observe that gobmk, hmmer, sjeng, and libquantum exhibit similar behavior in L1 TLB MPKI even with larger TLB configurations. The most demanding in terms of TLB miss rate is mcf, and even with the largest Configuration V the miss rate is still high.

TABLE III
NUMBER OF L2 TLB MISSES FOR MCF WITH 1024-ENTRY L2 TLB

L2 TLB Associativity	Direct-mapped	4-way	8-way
#TLB Misses for mcf	40.2M	6.9M	6.7M

Focusing on the impact of associativity, Table III shows the number of L2 TLB misses for mcf as we increase the L2 TLB associativity but keep the total number of L2 TLB entries constant. The L1 Instruction/Data TLB parameters are based on those of Configuration V. We observe that TLB misses reduce by 82.8% and 83.3% when associativity changes from direct-mapped to 4-way and 8-way, respectively. This behavior highlights the possible impact on the miss rate that a direct-mapped TLB may have due to conflicting entries, and the benefits of using a set-associative TLB. Note, however, that such

Fig. 3. Aggregated MPKI of the L1 Data/Instruction TLBs for the various TLB configurations.

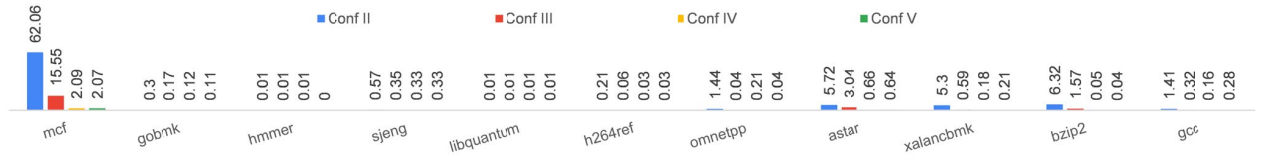

Fig. 4. Aggregated MPKI of the L2 TLB for the various TLB configurations.

TABLE IV
ABSOLUTE IPC VALUES FOR CONF. I AND PERCENTAGE OF IPC INCREASE FOR CONF. II TO V WITH RESPECT TO CONF. I.

Benchmark	I	II	III	IV	V
mcf	0.13	-	7.7 %	15.4 %	15.4 %
gobmk	0.44	-	-	2.3 %	2.3 %
hmmer	0.58	-	-	-	-
sjeng	0.55	1.8 %	1.8 %	1.8 %	3.6 %
libquantum	0.44	-	-	-	-
h264ref	0.77	1.4 %	1.4 %	2.6 %	2.6 %
omnetpp	0.35	2.9 %	5.7 %	5.7 %	5.7 %
astar	0.36	-	-	2.8 %	2.8 %
xalancbmk	0.36	2.8 %	8.3 %	8.3 %	8.3 %
bzip2	0.51	2.0 %	4.0 %	5.9 %	5.9 %
gcc	0.44	2.2 %	2.2 %	4.5 %	4.5 %

behavior depends on the working set of the application and its access pattern, and that our results are for the SPEC2006int benchmarks with the rather small test input set (Section IV).

Figure 4 shows the results of the MPKI in the L2 TLB for the various configurations. The Configuration I is not included, as it lacks an L2 TLB. We observe that the L2 TLB MPKI for most benchmarks is nearly zero, particularly for the larger Configurations IV and V, thanks to the larger reach of the L2 TLB. There is also a major improvement in mcf which stresses the most the L2 TLB. On average, the miss rate of L2 TLB is nearly zero with the larger Configurations IV and V.

Finally, Table IV summarizes the absolute IPC value with Configuration I, and the IPC speedup for Configurations II-V with respect to Configuration I. As we can see the IPC performance increases by up to 15.4 % depending on the demand of TLB resources and access patterns of every benchmark. In summary, our configurable TLB hierarchy can improve the performance with a slight impact in resource usage and nearly with no impact in frequency.

VI. Related Work

Prior work has focused on two major paths, (i) to improve specific soft-processor components to gain higher clock frequency and/or lower resource usage, and (ii) to bridge FPGA-to-ASIC performance to gain more insights about the actual performance of a processor to be fabricated and also lower resource usage. Our work is orthogonal to those approaches, as we design a configurable TLB hierarchy that is FPGA-friendly but can also be synthesized for ASICs, while obtaining actual performance results for various scenarios.

1) Improving soft-processor performance: There has been extensive prior work on improving soft-processor design [1]–[7] with hand-optimized HDL code and accurate micro-architectural design to produce as best as possible design mappings on an FPGA. Future work on Chisel HCL and FIRRTL Compiler [21] could target FPGA-specific structures to improve the overall performance of a design and reduce the resource utilization.

2) FPGA Resource Efficiency and Accurate Simulation: CAMs are known to be FPGA-hostile structures [22]. Magyar et al. proposed Golden Gate [23] to create Decoupled FPGA-accelerated Simulators by replacing CAMs with multi-cycle models, thus reducing resource utilization. As fully associative TLBs are typically implemented as CAMs, future work on resource optimization could aid FPGA-simulated research frameworks, especially on multi-core systems which have higher FPGA resource demand.

VII. Conclusions

In this paper we explored the Memory Management Unit of the Rocket Chip Generator and lifted its implementation limitations in the TLB hierarchy. We implemented a fully configurable L1 and L2 TLB, that can output any design from direct-mapped to fully-associative. Our approach enables design space exploration and allows the Rocket Chip Generator to instantiate cores with TLBs that match the needs of TLB intensive applications. We make our design publicly available to enable further research on the active topic of virtual memory support for the RISC-V architecture.

Acknowledgments

We would like to thank Dr. Tuo Li from the UNSW School of Computer Science and Engineering for porting the Rocket Chip Generator to the Xilinx ZCU102 board, his contribution and useful tips helped us considerably.

REFERENCES

[1] K. Aasaraai and A. Moshovos, "What limits the operating frequency of a soft processor design," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, 2014, pp. 1–6.

[2] K.Aasaraai and A. Moshovos, "An Efficient Non-blocking Data Cache for Soft Processors," in *2010 International Conference on Reconfigurable Computing and FPGAs*, 2010, pp. 19–24.

[3] M. Labrecque and J. G. Steffan, "Improving pipelined soft processors with multithreading," in *2007 International Conference on Field Programmable Logic and Applications*, 2007, pp. 210–215.

[4] D. Wu, K. Aasaraai, and A. Moshovos, "Low-cost, high-performance branch predictors for soft processors," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–6.

[5] E. Matthews and L. Shannon, "TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[6] H. Wong, V. Betz, and J. Rose, "Efficient methods for out-of-order load/store execution for high-performance soft processors," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 442–445.

[7] K. Aasaraai and A. Moshovos, "Design space exploration of instruction schedulers for out-of-order soft processors," in *2010 International Conference on Field-Programmable Technology*, 2010, pp. 385–388.

[8] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[9] Sifive. Freedom Platform. [Online]. Available: https://github.com/sifive/freedom

[10] Tuo Li. Rocket Chip Generator, Xilinx ZCU102 port. [Online]. Available: https://github.com/li3tuo4/rc-fpga-zcu

[11] UCB BAR. Support for Rocket Chip on Zynq FPGAs. [Online]. Available: https://github.com/ucb-bar/fpga-zynq

[12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, p. 1216–1225.

[13] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, p. 1–17, Sep. 2006.

[14] A. Waterman and K. Asanovic. (2017, May) The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.10. RISC-V Foundation.

[15] Andrew Waterman and K. Asanovic. (2017, May) The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2. RISC-V Foundation.

[16] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, Jun 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[17] Xilinx. Xilinx ZCU102 User Guide. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf

[18] Sifive. Freedom-U-SDK. [Online]. Available: https://github.com/sifive/freedom-u-sdk

[19] ARM. ARM Cortex-A57 Technical Reference Manual. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488c/DDI0488C_cortex_a57_mpcore_r1p0_trm.pdf

[20] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[21] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *Proceedings of the 36th International Conference on Computer-Aided Design*, 2017, p. 209–216.

[22] H. Wong, V. Betz, and J. Rose, "Quantifying the gap between fpga and custom cmos to aid microarchitectural design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2067–2080, 2014.

[23] A. Magyar, D. Biancolin, J. Koenig, S. Seshia, J. Bachrach, and K. Asanović, "Golden gate: Bridging the resource-efficiency gap between asics and fpga prototypes," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.