ANITA B.ORG

A Hardware Engine for Generating Number-Theoretic Sequences

Madeline Burbage
Williams College
3905 48th PL NE
Seattle, WA 98105
+1 (206) 602-5312
mgb4@williams.edu

1. INTRODUCTION

Number-theoretic sequences have been an indispensable component of hardware since the dawn of computing [3]. They play a part in hardware-level randomization, necessary for operations within a computer's architecture as well as for cryptography and communications. One useful class of sequences is combinatorial — bit patterns that illustrate different cases for choosing items from a set given certain constraints. Efficient algorithms for generating these sequences have been designed recently [5, 8]. Yet these software-implemented algorithms could be sped up further in hardware.

In this research we demonstrate the process of using open-source tools to create a combinatorial sequence generator with reconfigurable hardware [4]. Recent advances in hardware description and generation tools ease this process. Written in Chisel, a high-level hardware description language, our engine is implemented as a hardware accelerator for the RISC-V open-source processor. We then evaluate the tradeoffs of our hardware implementation.

2. RELATED WORK

Sequence-generating algorithms can be defined by a successor rule that creates each binary string from its predecessor. Stevens and Williams introduced three related rules, shown in figure 1, that cyclically generate certain combinatorial classes of binary strings using only simple rotations and bit flips [8]. Adjusting these rules can generate different weight ranges of binary strings, where the weight is the number of bits set to 1. Their first rule, the 'cool' order, visits all fixed-length strings with the same weight. The second, 'cooler' order, generates all possible strings of a certain length. Finally, the 'coolest' rule produces every string in a range of weights.

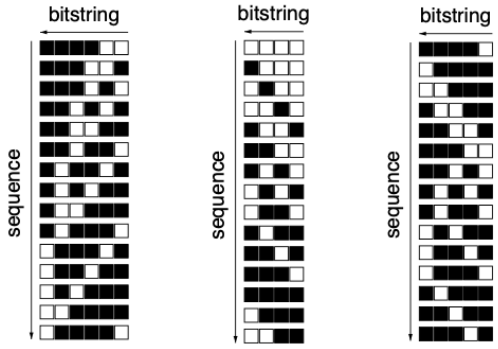

Figure 1. Combinatorial sequences of binary strings for the cool order with length 6 and weight 2, cooler with length 4, and coolest with length 5 and weights 1 and 2.

Knuth implemented the cool successor rule in MMIX assembly, in his survey of combinatorial algorithms [5]. Taking the length and weight of strings as input, the algorithm produces a full cycle of fixed-weight strings. His implementation uses clever bitwise manipulations to accomplish changes inexpensively.

We used Chisel to implement these algorithms in hardware. Chisel is a functional language extending Scala that brings modern programming practices to hardware generation [2]. Besides splitting complex hardware into modules generated by functions, Chisel allows programmers to build configurable hardware 'generators' [7]. Chisel can then produce circuitry to be simulated, mapped into FPGA chips, or directly fabricated in silicon, presenting robust options for testing and producing hardware.

To evaluate our sequence-generating engine, we connected it to the Rocket Chip – University of California, Berkeley's open-source processor

implementing the RISC-V open ISA [1]. The Rocket Chip's generator is written in Chisel, with many options for configuration. The chip also has interfaces which simplify the process of developing 'accelerators' for the processor [6]. The RoCC interface uses a custom instruction to link accelerators to the core and lets accelerators access the core's memory hierarchy.

3. APPROACH

3.1 Design

Using Knuth's approach, we implemented the three successor rules developed by Stevens and Williams into a Rocket Chip hardware accelerator. For each rule, our accelerator can either return the next string for a given string in a sequence or stream a full sequence of strings to memory. We've organized our accelerator into three sections: Command Control, Memory Control, and String Building (Figure 2).

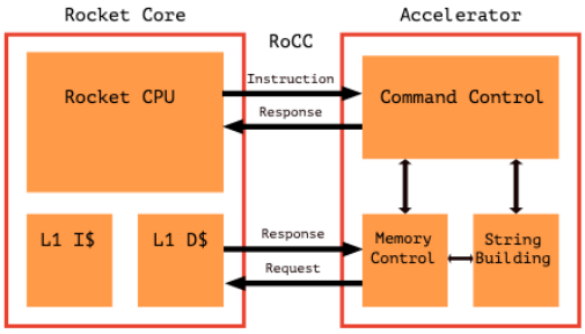

Figure 2. Accelerator Design.

The Command Control section is connected to the Rocket core and is responsible for receiving and responding to instructions through the RoCC interface. These instructions contain a function code dictating which sequence rule to use and the form of response, giving six total functions. The instruction also contains source registers with data that configures the sequence to be generated. If an immediate-response instruction is received, then the String Building unit produces the next valid string for the Command Control section to save to the destination register. Otherwise, the Memory Control section uses the String Building unit to generate a full orbit of valid bit patterns, storing them to the L1 cache.

Data stored from instructions is immediately used in the String Building section of the accelerator, which simultaneously calculates the next string for each successor rule to maximize output speed. Here we use the same bitwise operations as in the software version of the sequence generator. However, the operations are significantly sped up because the steps can be parallelized. This section computes changes to strings in less than a cycle, allowing the control units to coordinate result transfers every cycle.

When a function calls for strings to be stored to memory, the Memory Control unit sends storage requests through RoCC's memory interface and verifies their completion. The memory interface includes tags to allow for overlapping memory accesses. The Memory Control unit takes advantage of this, sending a request in every cycle that the cache is ready. Since string calculations are fast, the rate of memory storage is only slowed by the fact that one request can be sent each ready cycle.

3.2 Evaluation

We tested the speedup of the hardware implementations of the sequence generators by running the Rocket chip in a cycle-accurate simulator. The accelerator works for arbitrary bit pattern widths up to 32. However, the size of memory available to the simulator constrained our evaluation to patterns whose orbits generated fewer than 215 strings. We compared speedups for the three types of orders on strings of length 2, 4, 8. 16, and 24 for immediate-response functions, and 2, 4, 8, and 13 for memory-response functions to avoid storage overruns.

4. RESULTS AND CONTRIBUTIONS

4.1 Results

Figure 3 shows the results for immediate-response function speedup in hardware. While each function showed speedup, cool showed less than cooler, which showed less than coolest. These functions grow in complexity in that order, so their software versions grow in latency in that order as well. However, the hardware implementation finishes all calculations in one cycle, which explains why the more complex functions have greater speedup.

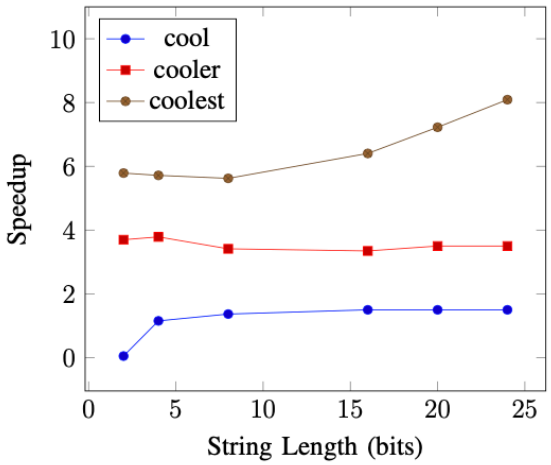

Figure 3. Speedup for immediate-response functions.

Additionally, our algorithms for cool and cooler run in constant time but coolest requires the use of a loop to check how many bits are set in the string, making it run in linear time. This explains why speedup for cool and cooler is constant but speedup grows with string length for coolest. We also noted that speedup is smaller for very small string lengths, because the software ran much faster than predicted. Optimized software skipped most calculations for small strings, explaining why the speedup is much smaller.

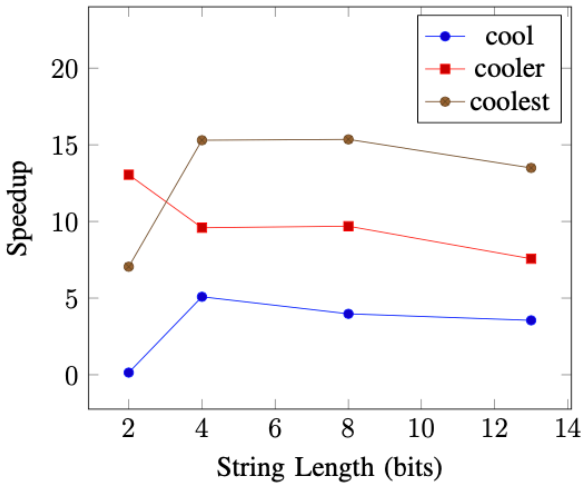

Figure 4. Speedup for memory response functions.

Figure 4 shows the results for speedup with memory-response functions, which is around twice as high as speedup for immediate-response functions. Since these functions remove the costly synchronization of hardware and software, they can stream strings to memory as fast as the cache can accept them.

Like with the immediate-response functions, shortest-length strings show less speedup, likely for the same software reasons. We also see the best speedup for medium-length strings, which can be explained by sequence orbit lengths. Latency here is driven by the cache's response times, which slowed or stalled when saving larger strings, likely because of their longer total sequence length. With medium-width strings there would be fewer stalls per orbit so the latency would be reduced most noticeably.

4.2 Contributions

The speedup found for every function type indicates the effectiveness of hardware acceleration for mathematically interesting successor functions. Our approach demonstrates the efficiency of using an open-source toolchain to accelerate specific software problems. We also see this work as a well-documented contribution to a growing collection of accelerators for RISC-V processors.

5. REFERENCES

[1] Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H., and Waterman, A. 2016. *The Rocket Chip Generator*. Technical Report. University of California, Berkeley.

[2] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawryzynek, J., and Asanović, K. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference* (June 2012). 1212-1221.

[3] Golomb, S. W. 2017. *Shift Register Sequences*, 3rd revised ed. World Scientific, 2017.

[4] Hauck, S., and DeHon, A. 2010. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation.* ISSN. Elsevier Science.

[5] Knuth, D. 2005. *The Art of Computer Programming, Volume 4, Fascicle 3.* Pearson Education.

[6] Mao, H. 2017. Hardware acceleration for memory to memory copies. Master's Thesis. University of California, Berkeley.

[7] Schoeberl, M. 2019. *Digital Design with Chisel.* Kindle Direct Publishing.

[8] Stevens, B., and Williams, A. 2014. The coolest way to generate binary strings. *Theory of Computing Systems* 54, 4 (May 2014), 551-557