View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Technische Universität Dresden: Qucosa

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

Technische Berichte
Technical Reports

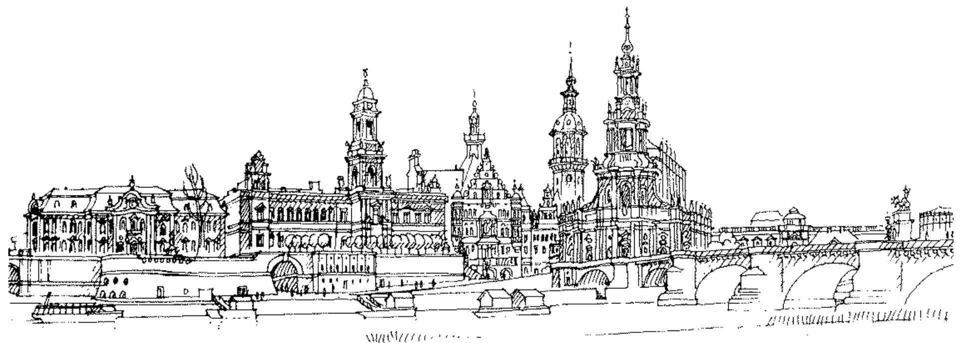
ISSN 1430-211X

TUD-FI06-03 September 2006

Thomas B. Preußer

Institut für Technische Informatik

Background of the Analysis of a Fully-Scalable Digital Fractional Clock Divider



Technische Universität Dresden
Fakultät Informatik
D–01062 Dresden
Germany
URL: http://www.inf.tu–dresden.de/

Background of the Analysis of a Fully-Scalable Digital Fractional Clock Divider

Thomas B. Preußer Technische Universität Dresden, Germany
Email: *preusser@ite.inf.tu-dresden.de*

***Abstract*— It was previously shown [1] that the BRESENHAM algorithm [2] is well-suited for digital fractional clock generation. Specifically, it proved to be the optimal approximation of a desired clock in terms of the switching edges provided by an available reference clock. Moreover, some synthesis results for hardwired dividers on Altera FPGAs showed that this technique for clock division achieves a high performance often at or close to the maximum frequency supported by the devices for moderate bit widths of up to 16 bits.**

This paper extends the investigations on the clock division by the BRESENHAM algorithm. It draws out the limits encountered by the existing implementation for both FPGA and VLSI realizations. A rather unconventional adoption of the carry-save representation combined with a soft-threshold comparison is proposed to circumvent these limitations. The resulting design is described and evaluated. Mathematically appealing results on the quality of the approximation achieved by this approach are presented. The underlying proofs and technical details are provided in the appendix.

I. INTRODUCTION

The BRESENHAM algorithm [2] is a long-known algorithm for the generation of plots of straight lines. It is the simplest and most fundamental representative of a class of incremental algorithms used for the efficient calculation of plots of curves on rastered devices based solely on fixed-point arithmetic. Others include the generation of plots for cyclic arcs [4] and elliptic curves [5].

The application of a hardware implementation of the BRESENHAM algorithm for fractional clock generation is mentioned in [6]. It is formally proven to be the optimal approximation of the desired clock in terms of the switching edges provided by an available reference clock in [1]. Latter work showed that a very straightforward implementation of a hardwired BRESENHAM clock divider performs well for moderate bit widths of up to 16 bits on current FPGA hardware. This paper extends on this work by showing the performance bounds of the direct implementation and by proposing an approach that trades some approximation quality for ideal scalability. It is shown that the incurred quality loss is fairly small and is not even present at all for most dividing fractions.

In the remainder of this paper: Sec. II reviews the BRESENHAM algorithm in the context of discrete fractional clock division. Sec. III proposes a design based on carry-save arithmetic and soft-threshold comparison. Its simulation and results about the quality of its generated clock are presented in Sec. IV. Sec. V concludes the paper and recapitulates the observations and open mathematical questions. The appendix, finally, contains the lengthier proofs for lemmas used in the paper as well as the source code for the simulative design quality evaluation.

Part of this work has been originally published in [3] copyrighted to and available from IEEE. Permissions for sale or commercial reproduction of this work are not granted.

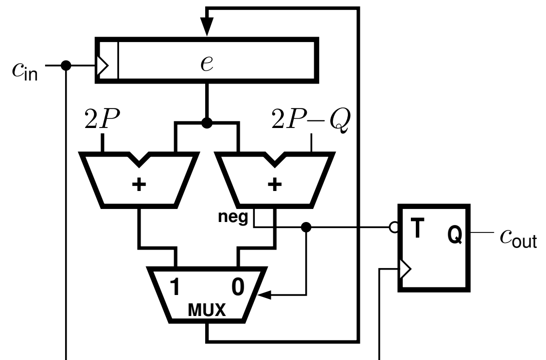


Fig. 1. Basic BRESENHAM Clock Division

II. REVIEW OF BRESENHAM CLOCK DIVISION

The hardware design proposed in [1] to implement the BRESENHAM clock division is reproduced in Fig. 1. The frequency of the generated clock is $f_{\text{out}} = \frac{P}{Q} f_{\text{in}}$. The output clock is free of any long-term phase drift. The initialization $e = Q - P$ was used in [1] to prove that the approximation of the ideal clock obtained by this design is optimal. All other initializations were shown to differ from this result merely in phase.

In [1] a few synthesis results for hardwired FPGA implementations were presented. The design is, however, equally applicable for a programmable VLSI block simply by turning the constant inputs $2P$ and $2P - Q$ into configuration registers.

For the further discussion, we will somewhat depart from the special case of clock division, which causes the appearance of $2P$ due to the need to generate two edges for each complete clock cycle of the output clock. In a more general discussion, p and q shall thus be used, which satisfy $p = 2P$ and $q = Q$ for the clock division. Recall the requirement $q \geq p$ that carries over from the original line drawing application.

The operation implemented by each iteration is the addition of p modulo q so that e iterates through remainder classes of q using representatives from $[0, q)$. This choice certainly minimizes the bit width used in the design but is nevertheless arbitrary. In fact, a whole adder can be eliminated if the case when $p-q$ instead of p is to be added – call it the *modulo event* – is identified by the value of e rather than by the sign of a speculative addition. This can be achieved either (a) by choosing representatives in a range $[2^{n-1}+p-q, 2^{n-1}+p)$ triggering the modulo event by the most significant bit (MSB) of e with value 2^{n-1} or, similarly, (b) by subtracting instead of adding p and $p-q$ combined with the sign detection of the two's complement representation of e, again by its MSB. Latter approach gives e a range of $[-p, q-p)$.

The required bit widths for these implementations are as follows:

(a) Since the MSB triggers the modulo event when the representative assumes at least the value 2^{n-1}, all representatives must be positive to avoid a false trigger. This yields:

$$\begin{array}{lll} 2^{n-1}+p-q & \geq & 0 \\ 2^{n-1} & \geq & q-p \\ n-1 & \geq & \mathrm{ld}(q-p) \\ n & \geq & 1+\mathrm{ld}(q-p) \end{array}$$

Further, $(2^{n-1}-1)+p$ must be representable in n bits. Thus:

$$\begin{array}{lll} 2^{n} & \geq & 2^{n-1}+p \\ 2^{n-1} & \geq & p \\ n-1 & \geq & \mathrm{ld}\,p \\ n & \geq & 1+\mathrm{ld}\,p \end{array}$$

Joining both conditions yields:

$$\underline{\underline{n \quad \geq \quad 1+\mathrm{ld}\max\{q-p,\, p\}}}$$

(b) This case requires that the two's complement representation of n bits covers the whole range $[-p, q-p)$. Thus, $-2^{n-1} \leq -p$ and $q-p \leq 2^{n-1}$ need to be satisfied, which yields exactly the same bound as case (a).

Therefore, both of these cases will never require a smaller bit width than the original implementation demanding a minimum bit width of $\mathrm{ld}\,q$. Nonetheless, they also require at most one bit more. Unless, the carry propagation delay in the adder is absolutely tight, this is definitely a good tradeoff. The resulting design, here for case (a), is depicted in Fig. 2.

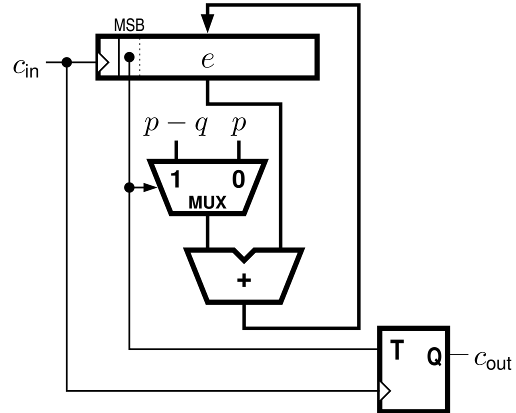


Fig. 2. Design (a) with Alternate Modulo Class Representatives

The performance-limiting factor of the designs seen so far is the carry propagation within the adder. Even when choosing fast binary adder implementations, the achievable combinatorial delay is still $\Omega(\log n)$ [7]. Optimizations for the original line drawing application cannot be transfered to the clock division as they rely on parallelization and / or the identification of identical line segments [8]. Faster implementations in the clock division domain are only possible by the elimination of the carry propagation. If such can be found, their adoption for line drawing is very well possible.

III. A Fast, Hardly Approximating Design

Addition can be performed faster when the redundant representation of e is permissible. In fact, the coding of e is arbitrary as long as the modulo events adding $p-q$ instead of p can be identified. So far, firm thresholds (q, 2^{n-1} or 0) were used to trigger these events but unfortunately:

Lemma 1: Be G a totally-ordered group and $+ : G \times G \rightarrow G$ the group operation called addition. Then, there is no representation for the members of G, which allows both the addition and the comparison against some fixed group member to be implemented faster than $\Omega(\log\log|G|)$.

Proof: Assume there was such a representation. Be t the group member that can be compared against faster than $\Omega(\log\log|G|)$. So an input x can be identified exactly as t by testing both $x < t$ and $t < x$ (or $x \leq t$ and $t \leq x$). Further, a circuit adding $x + d$ and testing the result for equality with t can identify any group member a by setting $d = -a + t$. As this circuit can be programmed to identify any group member a, it must have $w = \lceil \log_v |G| \rceil$ input lines for x with v possible input values per line to code all group members uniquely. The reduction of these w input lines to a single signal

Fig. 3. Carry-Save Digital Clock Division Design

indicating equality or no takes, at least, $\lceil \log_r w \rceil$ steps when implemented using gates with at most r inputs (also see [7]). As additon combined with comparison thus is of $\Omega(\log \log |G|)$, not both of these operations can be of lower order at the same time. ■

Applied to addition, this means that not both a fast addition and a fast comparison against a firm threshold are possible at the same time. In terms of the bit width n, $\Omega(\log n)$ is the best to achieve. A way out of this dilemma is the soft-threshold comparison.

Consider the design given in Fig. 3. It resembles the design from Fig. 2 except for the representation of e and the identification of the modulo event. e is encoded in carry-save representation, i.e. its numerical value is the sum of two pseudo-components – e^s, the pseudo-sum, and e^c, the pseudo-carry. The modulo event is triggered, depending on the configuration bit t, by a single or by two bits set in the MSB position. The performance of this design is no longer limited by a carry propagation path. Its critical combinatorial path is of $O(1)$. Only, the signal controlling the wide $p/p-q$-MUX is loaded heavily and thus likely to require some effort in a practical design.

The quality of the clock generated by this design is not clear at all. Due to the redundant nature of the carry-save representation, certain values of e may trigger the modulo event when represented one way and may not when represented another. Thus, the series of values in e is not predetermined by design and might not only depend on the choices of p, q and t but also on the initial value and representation of e. Yet, the modulo classes with respect to q that are represented by the values of e succeed in a well-defined order as the additions of p or $p-q$ are equivalent in terms of these classes.

As the number of states that e can assume is finite, a cycle must be entered eventually. As the equal states of e in successive iterations of this cycle represent the same modulo class, the period of any such cycle is a multiple of q (or $\frac{q}{\mathrm{gcf}(p,q)}$ if p and q are not relatively prime). Since the additions of p and $p-q$ also sum up to zero over a whole cycle, their overall ratio equals that of the original BRESENHAM clock division. Their distribution, however, may differ so that the generated modulo events no longer constitute the best approximation of the desired clock at the output. So the quality of the output of the proposed design will need to be evaluated.

For the design to work, it is absolutly vital that the additions do not produce an arithmetic overflow. For a width of n bits, an overflow would essentially cause a $\bmod 2^n$-operation, which will only go consistent with the calculation within the modulo classes of q in the special case that $q = 2^{n-k}$. So it must be ensured

1) that the addition of p never produces an outgoing carry and
2) that the addition of $p-q$, which is actually a subtraction, always produces an outgoing carry.

The first requirement implies that, if both MSBs of e are set, the modulo event must be triggered. The second requirement implies that the modulo event must not be triggered if not at least one of the MSBs of e is set. As these border cases are easily identified by the inspection of only two bits, they were chosen as the two options to

investigate. In the design, they can be configured via t.

Observe that the use of the carry-save representation slightly differs from its application in common arithmetic settings. Here, the numeric value of e is strictly defined to be the positive sum of both unsigned pseudo-components – without any modulo operation. Specifically, the n-bit-wide carry-save representation with $e^s = 2^n - 1$ and $e^c = 1$ is not another representation for the value 0 but represents $e = 2^n$. Further, note that the application of the carry-save representation in the proposed design allows the formation of equivalence classes among the representations of an integer e. Interpreting the two hardware bits at each bit position as encoding one of the digits $\{0, 1, 2\}$, the two different encodings of the digit 1 are not distinguished by the carry-save adder and can thus be considered equivalent. So the two representations $9+3 = \binom{1001}{0011}$ and $11+1 = \binom{1011}{0001}$ for 12 are equivalent as both recode to 1012_2. $7 + 5 = \binom{0111}{0101}$, on the other hand, recodes to 0212_2, thus not being equivalent to those representations.

Definition 1: A representation of a natural number in the redundant place-value system with the digits $\{0, 1, 2\}$ and the base 2 is called *additive carry-save* representation.

As the additive carry-save representation is predominant in this paper, all references to carry-save shall mean additive carry-save unless explicitly stated otherwise.

The range of values traversed by e during a cycle is no longer strictly confined to an interval of length q. The smallest possible value that e can assume within a cycle is reached after the smallest e that has a carry-save representation triggering the modulo event; the largest possible value of e within a cycle is reached after the greatest e that has a carry-save representation not triggering the modulo event. Thus, the *cyclic values* of e lie somewhere in:

$$\begin{array}{ll} [2^{n-1} + p - q,\ 2 \cdot (2^{n-1} - 1) \quad -1 + p] & \text{if } t = 0 \\ [2^{n} \quad + p - q,\ (2^n - 1) + (2^{n-1} - 1) - 1 + p] & \text{if } t = 1 \end{array}$$

These intervals be called the *cyclic range* E of e.

For the determination of the minimum implementation bit width n, consider the following requirements:

- the negative number $p - q$ must be representable as a two's complement of n bits,
- p must be representable as an unsigned natural of n bits; for $t = 1$, it may not have a set MSB so as not to provoke an outgoing carry from the CSA, and
- the cyclic range of e must be representable in additive carry-save with components at most n bits wide.

The intersection of these requirements yields:

$$n \geq \begin{cases} \max\{\quad \mathrm{ld}(p+1), 1 + \mathrm{ld}(q-p)\} & \text{if } t = 0 \\ \max\{1 + \mathrm{ld}(p+1), 1 + \mathrm{ld}(q-p)\} & \text{if } t = 1 \end{cases}$$

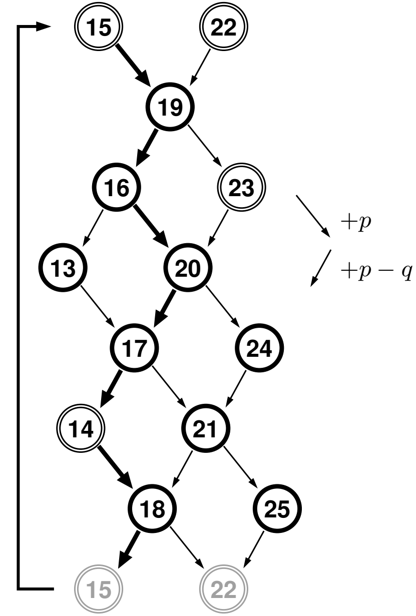


Fig. 4. Lattice of e-Values for (p,q;n,t)=(4,7;4,1)

These bounds are fairly similar to the one found for the modified original design of Fig. 2. So the main investment into the carry-save implementation is the coding overhead for the representation of e doubling the register size. There is no significant gain or price paid in a larger bit width. Multiplexer and adder thus essentially have the same complexity.

The members of the cyclic range E can be organized in a *lattice*-like graph. (Note that the term lattice is merely inspired by the shape and has no relation to the algebraic lattice established by certain posets.) An example of such a lattice for $(p, q) = (4, 7)$ with $n = 4$ and $t = 1$ is given in Fig. 4. Each edge of this graph resembles a transition from one value of e to another. An edge to the left implies that the value of the source node has a carry-save representation triggering the modulo event; an edge to the right analogously implies that it can be represented in a way that the modulo event is not triggered. Note that many values of e have carry-save representations of both kinds. Further, observe that the lattice has been drawn such that horizontally neighboring nodes are two values representing the same modulo class with respect to q.

As it turns out, the cycle for the case examplified in Fig. 4 is unique and has a period of exactly q. For the argument of uniqueness, observe that the carry-save representations of $2^n - 1$ and $(2^n - 1) + 2^{n-1}$ (here 15 and 23) are unique, i.e. all equivalent according to the discussion above. Further, the lack of an incoming carry to the carry-save adder restricts the value of the least significant digit of the sum to 0 or 1; it cannot be 2. Thus, also $(2^n - 1) - 1$ and $((2^n - 1) + 2^{n-1}) - 1$ (here

14 and 22) have unique representations since they are even, which implies the least significant digit is 0, and by truncating this 0 they become unique representations of the bit width $n-1$ ($2^{n-1}-1$ and $2^{n-1}-1+2^{n-2}$). As every cycle must include a representative of any one module class and both cyclic representatives of $15, 22$ are uniquely represented, it is easy to verify that every cycle must pass through the same representation of 19 (which is 2011_2). This, in turn, implies a unique and primitive cycle of period q. Its simulation yields the path boldened in Fig. 4. Most importantly, the distribution of the modulo events on this path is equivalent to the one produced by the original BRESENHAM clock division design.

Unfortunately, this advantageous behavior does not generalize. Simulation shows, however, that it is astonishingly common. Moreso, the quotients $\frac{p}{q}$ that do not yield a full-quality BRESENHAM-like sequence of modulo events seem to cluster in well-defined, diamond-shaped areas on the p-q-plane. These results are described in detail in the following section.

IV. SIMULATION AND RESULTS

The goal of the simulation was to obtain some information about the cyclic behavior of the proposed design:

- How many cycles are there?
- What are their periods?
- How well do they approximate the desired output clock also as compared to the original BRESENHAM clock division.

While the knowledge about the approximation quality is an obvious goal, the others may require some explanation. The number of the cycles the design may possibly enter is valuable as a unique cycle may serve to relax the initialization requirements. If it is sufficient for a design to output the promised quality eventually, no explicit initialization is required at all. Knowledge about the period of the cycles may be a first step to a provable quality assurance as no multi-q-cycle can produce a better quality than the primitive BRESENHAM cycle with period q.

The simulation must be restricted to some attractive range. So the bit width n used in the simulation was limited to its minimum $\check{n}$ and one bit wider. Note that solutions found for some bit width can be easily scaled to larger bit widths by shifting the involved constants p and $p-q$ as well as the initialization of e left by the number of additional bits. No output quality is lost by this scaling.

The quotients $\frac{p}{q}$ simulated were restricted to reduced fractions where p and q are relatively prime. This ensures that every cycle of e must include a representative of any one modulo class of q. Thus, the set of initial values of e that need to be considered to identify all cycles can be limited to all representatives of exactly one of these modulo classes from the cyclic range E.

Also, the cycle identification can be based purely on the representatives of this modulo class. As if establishing checkpoints at a horizontal cut through the lattice of e-values, only the e reached after every q steps needs to be compared against previously encountered states. Also remembering the states of e reached by a simulation run with another initialization of e helps to further reduce the simulation effort as the cycle reached from this point is already known so that the simulation for the current initialization can be quit.

The set of initial states of e that need to be simulated to identify all cycles comprises all additive carry-save representations of all representatives of the chosen modulo class. Avoiding an explicit search for the smallest of these sets, the modulo class containing 2^n-1 was chosen as at least this representative has a unique carry-save representation. All other $s(k) = 2^n - 1 + k \cdot q \in E$ ($k \in \mathbb{Z}$) for $k \neq 0$ may have several carry-save representations. These can be generated by a sliding additive partition, which may, however, produce duplicates only differing in their coding of 1-valued digits. In order to reduce the simulation effort, such duplicates need to be identified and eliminated. Their identification can be based on a simple bitwise XOR of their pseudo-components, which essentially identifies the distribution of their 1-valued digits:

Lemma 2: All additive carry-save representations (and traditional carry-save representations when the encoding of 1-valued digits is irrelevant) of an integer e are uniquely identified by the distribution of their 1-valued digits.

Proof: (*by Contradiction*) Assume there would be two carry-save representaions of e with the same distribution of 1-valued digits. The numerical value of only these 1-valued digits be e'. Since e has two different carry-save representations, so has the difference $e-e'$ simply by substituting a 0 for every 1 in both representations of e. By this construction, both representations of $e-e'$ are only comprised of the digits 0 and 2. Substituting a 1 for every 2 in these representations yields two different, now conventional binary representations of the value $\frac{e-e'}{2}$ $\rightarrow$ a contradiction. ∎

For the estimation of the simulation effort for a single reduced fraction $\frac{p}{q}$, it is valuable to know the bounds on

- the number of representatives of the modulo class chosen for the initial values within the cyclic range E,
- the number of their distinct carry-save representations, and
- the simulation effort necessary for each of these valid initializations.

The early dropout from all simulation runs, as soon as e reaches a state already encountered before, limits the

amortized effort spent on each initialization to the q steps constituting one complete up-down traversal of the e-lattice. The number of distinct carry-save representations for each initial value $s(k)$ is, according to Lemma 3 in the appendix, of $O\left(s(k)^a\right)$ with $a = \operatorname{ld}\left(\frac{1+\sqrt{5}}{2}\right)$. Applying the restriction that only the minimum value of n and its successor are simulated, this reduces to $O(q^a)$. The number of representatives of the chosen modulo class within E grows with $O\left(\frac{2^n}{q}\right)$, which reduces to a small constant by the limited choices of n. Thus, the overall simulation effort for a single reduced fraction $\frac{p}{q}$ is of $O\left(q^{1+a}\right) = O\left(q^{1.694242}\right)$.

The quality of the generated output is to be evaluated. Using a cycle of the input clock as time unit, a suitable measure can be obtained by the average distance square of the integer times of the modulo events from their optimal occurence on the continuous real timescale. As the evaluation is to reflect the edge jitter rather than the phase of the output clock, an arbitrary phase is allowed to minimize this deviation.

The evaluation of the original BRESENHAM approximation provides a baseline for comparison. As established by Lemma 4 in the appendix, the average distance error of this implementation can be explicitly given by $\frac{1}{12} \cdot \left(1 - \frac{1}{p^2}\right)$ for the approximation of a fraction $\frac{p}{q}$. Note that this term only depends on the numerator of the fraction and is always smaller than $\frac{1}{12}$, which establishes the asymptotic bound for growing p.

The simulation for all reduced fractions $\frac{p}{q}$ with $1 \leq p < q \leq 4096$ was implemented in Java [9]. Its results permit the following observations:

- The e-cycles of the implementations of the minimum bit width $\check{n}$ are unique (there is exactly one cycle) and primitive (they have the minimum period of q) for both choices of t.
- Cycles of the implementations of bit width $\check{n}+1$ are neither necessarily unique nor necessarily primitive.
  - The smallest example yielding two cycles is $(p, q; n, t) = (2, 3; 4, 1)$ with primitive cycles through 1210_2 and 2101_2.
  - The smallest examples having a unique cycle with period $2q$ is $(p, q; n, t) = (2, 5; 4, *)$.
- Although there is a great similarity between the results achieved for a single bit width n for both choices of t, no two of the four simulated settings yield the exact same quality result for all $\frac{p}{q}$. Moreso, none of the settings is better than some other for all fractions.
- The results achieved for the minimal bit width $\check{n}$ are usually at least as good as the ones achieved for a bit width of $\check{n} + 1$. The smallest exception to this rule is $\frac{6}{17}$ where the wider implementations achieve original BRESENHAM quality while the minimum ones do not.
- Original BRESENHAM quality is achievable for most fractions in some of the settings $(n, t) \in \{\check{n}, \check{n} + 1\} \times \{0, 1\}$. The smallest counterexample where none achieves this quality is $\frac{6}{13}$.

Note that all these observations are no proven universal statements yet. So there are quite a few open questions.

Most interestingly, the fractions $\frac{p}{q}$ that are the exceptions to the rules seem to cluster on or in diamond-shaped areas around some $p = m \cdot q$ with $p = 2q$ being the most dominant. This does not appear to be a simple consequence from the calculation of the minimum bit width although the term relevant for the maximum switches about this line. Similar patterns can be observed when n is, for example, left constant for a single q as by using $n = 1 + \operatorname{ld} q$.

To appreciate the structure apparently inherent to the problem, have a look at Fig. 5. Note that the quality charts show a normalized quality measure obtained as the quotient of the average error square of the orginal BRESENHAM quality divided by that achieved by the specified setting of the proposed design. Thus, shaded areas represent those fractions, for which the original quality cannot be totally achieved.

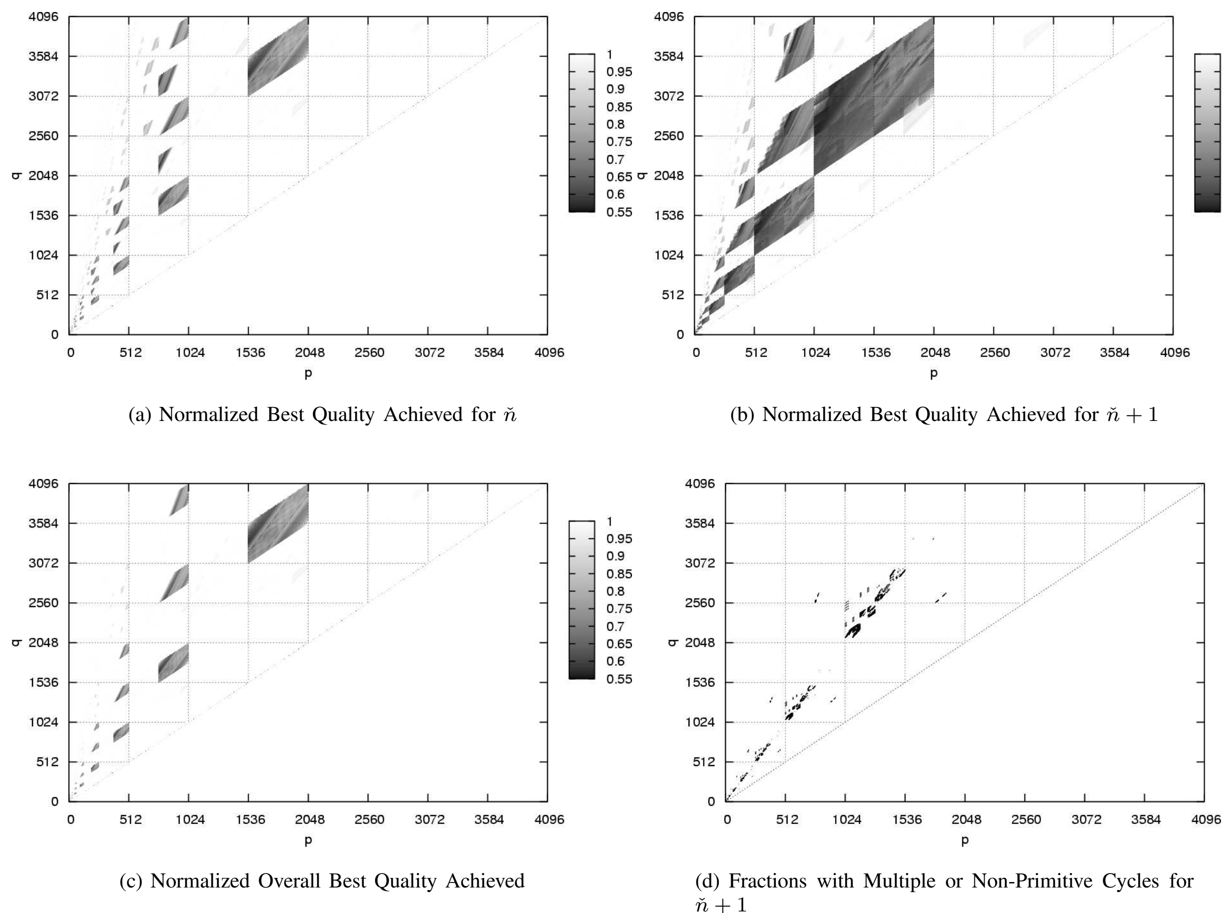
One is compelled to assume a regular repetitive pattern in the plotted graphs. Although there is no reason to believe that this observation is not to generalize, the inherent nature of their formation could not yet be discovered not to mention formally proven. Assuming that a generalization was valid, slightly more than 93% of all fractional divisions could be approximated with the same quality as by the original BRESENHAM design – already by one of the four investigated setups of the proposed design.

V. CONCLUSIONS

Optimizations of the straightforward BRESENHAM implementation for discrete fractional clock division have been discussed. As only little could be achieved for the original design, a further approximation step introducing a soft-threshold comparison to implement the modulo addition has been proposed. This approach enabled a highly-scalable design with a critical combinatorial path independent from the bit width and only one heavily-loaded logic signal.

The simulation of this design for fractions $\frac{p}{q}$ with relatively small p and q suggested that the proposed design achieves a high-quality clock output, for most fractions even equivalent to the quality achieved by the original BRESENHAM design.

Taking the required implementation bit width and the choice of the two easily-identifyable thresholds as parameters, no setting proved superior for all fractions although, in most cases, the choice of the smallest allowable bit width already yields the best results. Due



(a) Normalized Best Quality Achieved for $\check{n}$

(b) Normalized Best Quality Achieved for $\check{n} + 1$

(c) Normalized Overall Best Quality Achieved

(d) Fractions with Multiple or Non-Primitive Cycles for $\check{n} + 1$

Fig. 5. Visualized Simulation Results

to the observed uniqueness of the cycles in this minimal setting, it may be used as the base of a programmable implementation without extensive initialization logic required to ensure the entering of the correct high-quality cycle.

Quite a few questions are raised by the simulation results, which are still to be answered. Most importantly:

- Are all cycles for the minimum implementation bit width unique and primitive?
- What choice of the parameters n and t yields the best result for a reduced fraction $\frac{p}{q}$?
- Can better results be obtained by allowing expanded fractions?
- Can the original BRESENHAM quality be achieved for every fraction?
- Can a best-quality setup of the proposed design be identified efficiently?

Another interesting question that will hopefully be answered by the work on the others is: What makes the fractions within those diamond-shaped areas so special to regularly be the exceptions to the rules?

REFERENCES

[1] T. B. Preußer and S. Köhler, “Discrete fractional clock generation for systems-on-fpga,” Fakultät Informatik, Technische Universität Dresden, Tech. Rep. TUD-FI05-07, June 2005, ftp://ftp.inf.tu-dresden.de/pub/berichte/tud05-07.pdf.

[2] J. E. Bresenham, “Algorithm for computer control of a digital plotter,” *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

[3] T. B. Preußer and R. G. Spallek, “Analysis of a fully-scalable digital fractional clock divider,” in *ASAP 2006 – Application-specific Systems, Architectures and Processors*, Sept. 2006.

[4] J. Bresenham, “A linear algorithm for incremental digital display of circular arcs,” *Commun. ACM*, vol. 20, no. 2, pp. 100–106, 1977.

[5] M. L. V. Pitteway, “Algorithm for drawing ellipses or hyperbolae with a digital plotter,” *Computer Journal*, vol. 10, no. 3, pp. 282–289, Nov. 1967.

[6] *3.3V AnyClock Fractional N Synthesizer*, Micrel, Inc., 1849 Fortune Drive, San Jos’e, CA 95131, USA, http://www.micrel.com/_PDF/HBW/sy87729l.pdf.

[7] S. Winograd, “On the time required to perform addition,” *J. ACM*, vol. 12, no. 2, pp. 277–285, 1965.

[8] E. Angel and D. Morrison, “Speeding up bresenham’s algorithm,” *IEEE Computer Graphics and Applications*, vol. 11, no. 6, pp. 16–17, Nov. 1991.

[9] “Java technology,” Sun Microsystems, Inc., http://java.sun.com/.

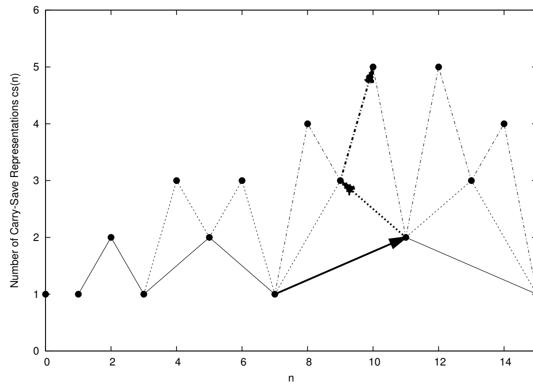


Fig. 6. Pattern Repetition in $\mathrm{cs}(n)$ in $[0, 15]$

Appendix

A. Number of Distinct Additive Carry-Save Representations

Lemma 3: The number of distinct additive carry-save representations of a natural number n is $O(n^a)$ where a is the dual logarithm of the golden ratio Φ: $a = \mathrm{ld}\left(\frac{1+\sqrt{5}}{2}\right) \approx 0.694242$.

Proof: First, observe that the representation of 0 is unique.

Now, consider an arbitrary odd number $n = 2k + 1$ ($k \in \mathbb{N}$). The least-significant digits of all its carry-save representations must be 1. Each of these representations corresponds 1-to-1 to a representation of k by truncating this 1.

Finally, consider an arbitrary even number $n = 2k$ ($k \in \mathbb{N}$). The least-significant digits of all its carry-save representations are either 0 or 2:

- Each representation ending in 0 corresponds 1-to-1 to a representation of $2k + 1$ by substituting this 0 for a 1.
- Each representation ending in 2 corresponds 1-to-1 to a representation of $2k - 1$ by substituting this 2 for a 1.

Thus, calling the function mapping a natural n to the number of its distinct additive carry-save representations cs, conclude:

$$\mathrm{cs}: \quad \mathbb{N} \to \mathbb{N}$$
$$0 \mapsto 1 \tag{1}$$
$$2k+1 \mapsto \mathrm{cs}(k) \tag{2}$$
$$2k \mapsto \mathrm{cs}(2k-1) + \mathrm{cs}(2k+1) \tag{3}$$

Due to (2) the pattern of the functional graph is copied from each interval $(2^{t-1} - 1, 2^t - 1]$ to the odd numbers of the succeeding interval $(2^t - 1, 2^{t+1} - 1]$. The even numbers in latter interval are filled by (3). This is illustrated in Fig. 6 for $0 \leq t \leq 3$.

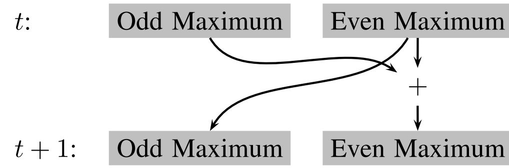


Fig. 7. Maximum Transition

The intervals $t : (2^{t-1} - 1, 2^t - 1]$ are chosen such that the left exclusive and the right inclusive border are odd and have unique carry-save representations, a property that carries from one interval to its successor. Starting with $t = 2$, the intervals span at least across two integers. Maxima in these intervals must be at even positions as, due to (3) and $\forall n \in \mathbb{N}.\, \mathrm{cs}(n) \geq 1$, both even neighbors of an odd position have greater values.

Call a maximum value of an interval at an odd position, its odd maximum; a maximum value at an even position, which corresponds to an interval-wide maximum, an even maximum. It can be established for $t = 3 : (3, 7]$ that the odd maximum $(5, 2)$ is located next to the even maxima $(4, 3)$ and $(6, 3)$.

Assume odd and even maxima neighbor each other in an interval $t : (2^{t-1} - 1, 2^t - 1]$. Copying to the succeeding interval $t + 1$ places them at neighboring odd positions according to (2). Since no other formerly odd position can have a larger value than the former odd maximum and no other formerly even position can have a larger value than the former even maximum, the even position in the succeeding interval between the neighboring former odd and even maxima becomes a new even and interval-wide maximum. This even maximum is again neighbored by an odd maximum, the former even maximum. Thus, even and odd maxima of an interval are always direct neighbors. Furthermore, a new interval-wide maximum is essentially the sum of the maxima from the two proceeding intervals. This results in the maxima to establish the Fibonacci series:

$$y(t) = F_{t+1} \tag{4}$$

Starting with $(3, 7]$, each interval has two maxima. The relative position of the left of these interval maxima can be determined by tracking the path established by the copies of previous maxima as depicted by the boldened path in Fig. 6 for $t = 4$. It is given by the partial sums of the alternating geometric series $\frac{1}{2} - \frac{1}{4} + \frac{1}{8} - + \ldots$.

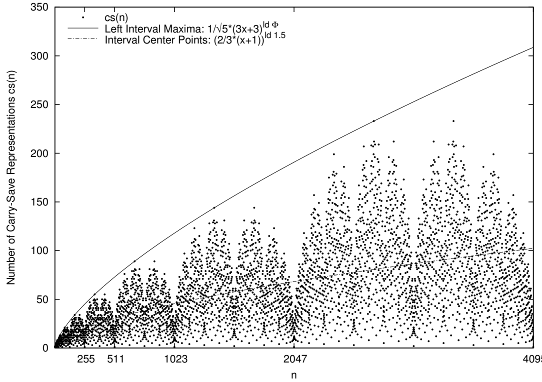


Fig. 8. $cs(n)$ with $O(x^a)$-Bound and Interval Center Curve

For large t this value approaches towards:

$$\begin{aligned} \frac{1}{2}-\frac{1}{4}+\frac{1}{8}-+\ldots &= \frac{1}{2}\sum_{i=0}^{\infty}\left(-\frac{1}{2}\right)^{i} \\ &= \frac{1}{2}\cdot\frac{1}{1+\frac{1}{2}} \\ &= \frac{1}{3} \end{aligned}$$

The absolute position of the left interval maxima thus approaches for large t:

$$\begin{aligned} x(t) &= \frac{2}{3}\cdot\left(2^{t-1}-1\right)+\frac{1}{3}\cdot\left(2^{t}-1\right) \\ &= \frac{2^{t+1}}{3}-1 \end{aligned} \tag{5}$$

Further, BINET's formula for the FIBONACCI numbers:

$$\begin{aligned} F_k &= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{k}-\left(\frac{1-\sqrt{5}}{2}\right)^{k}\right) \\ &= \frac{1}{\sqrt{5}}\Phi^{k}+\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{k} \end{aligned}$$

applied on (4) while ignoring the second addend approaching zero for growing k, a parametric description of the locations of the left interval maxima is obtained together with (5). This can be generalized to a function over all non-negative reals and transformed into the explicit form:

$$\begin{aligned} y(x) &= \frac{1}{\sqrt{5}}\cdot\Phi^{\operatorname{ld}(3x+3)} \\ &= \frac{1}{\sqrt{5}}\cdot(3x+3)^{\operatorname{ld}\Phi} \end{aligned} \tag{6}$$

This function is concave and differentiable. As all interval maxima of cs lie with a diminishing error about this function, it can be concluded that

$$\operatorname{cs}(n)\in O\left(n^{a}\right)\ \text{ with } a=\operatorname{ld}\left(\frac{1+\sqrt{5}}{2}\right)$$

■

Observe that the average value of cs within an interval grows marginally slower – not only by another constant coefficient but a slightly smaller exponent. While the sum of the values of cs triples from one interval to its successor (the odd positions are copied and they contribute twice to the values of their even neighbors), the width of the interval is only doubled. Verifying the start condition for $t = 1$, the average value of cs in the interval $t : (2^{t-1}-1, 2^{t}-1]$ is $y(t) = \left(\frac{3}{2}\right)^{t-1}$. Combined with the center position of the symmetric intervals $x(t) = \frac{\left(2^{t-1}-1\right)+\left(2^{t}-1\right)}{2} = 3\cdot 2^{t-2}-1$, the explicit form $y(x) = \left(\frac{2}{3}(x+1)\right)^{\operatorname{ld}\frac{3}{2}}$ can be obtained. The central points of the intervals thus lay on a curve only growing with $\Theta(x^a)$ with $a = \operatorname{ld}\frac{3}{2} \approx 0.584963$.

B. *Error of the Original* BRESENHAM *Approximation*

Lemma 4: The average square error of the times of the modulo events generated by the original BRESENHAM implementation for a fraction $\frac{p}{q}$ from their ideal occurence on a continuous time scale is $\frac{1}{12}\left(1-\frac{1}{p^2}\right)$ when taking a full clock cycle of the reference clock as time unit and neglecting the phase.

Proof: Assume the first modulo event at time $t_0 = 0$ starts a period with zero error to its ideal occurence. All p ideally aligned events of this period would have to occur at $t_i = i\frac{q}{p}$ with $i \in [0, p)$. As shown in [1], the approximated events generated by the original BRESENHAM implementation occur at times $T_i = \left\lceil i\frac{q}{p}-\frac{1}{2}\right\rceil$.

The error incurred by the first approximation is $\varepsilon_1 = T_1 - t_1 = T_1 - \frac{q}{p}$ where T_1 is an integer and thus $\varepsilon_1 = \frac{k}{p}$ with $k \in \mathbb{Z}$. The following errors ε_i are essentially multiples of ε_1 such that $\varepsilon_i = \frac{(ik)}{p}$. Due to the rounding to the nearest integer, (ik) is that representative of the modulo class of ik with respect to p, which has the smallest absolute value (positive on a tie but this is irrelevant for the result). Since q and p are relatively prime so are k and p. Thus, ik iterates through all p modulo classes of p within one period. So each of the errors $\varepsilon = \frac{j}{p}$ with $j \in \mathbb{Z}\cap\left(-\frac{p}{2}, \frac{p}{2}\right]$ occurs exactly once.

As the phase of the generated clock is to be neglected, an arbitrary but constant offset d to T_i is allowed to improve the overall approximation quality. Given the result above the overall sum of the quadratic errors within a complete period can be given as:

$$s^2 = \sum_{i=-\left\lfloor\frac{p-1}{2}\right\rfloor}^{\left\lfloor\frac{p}{2}\right\rfloor}\left(\frac{i}{p}-d\right)^2$$

For odd p, this reduces to:

$$s^2 = p\cdot d^2+\frac{p^2-1}{12p}$$

This term is obviously minimal for $d = 0$, i.e. without a phase offset.
For even p, the following expression is obtained:

$$s^2 = p \cdot d^2 - d + \frac{p^2 - 3p + 2}{12p} + \frac{1}{4}$$

This term assumes its minimum for a phase offset of $d = \frac{1}{2p}$.
Both cases yield for the minimum:

$$s^2 = \frac{p^2 - 1}{12p}$$

Averaging over all p approximations of a period yields the desired term:

$$\frac{s^2}{p} = \frac{p^2 - 1}{12p^2} = \frac{1}{12}\left(1 - \frac{1}{p^2}\right)$$

■

Thus, the average quadratic error approaches $\frac{1}{12}$ for growing p. Smaller p achieve better results. Perfect approximations are only achieved for $p = 1$. Most notably, the approximation quality is independent from q.

Note that this result only applies to fully reduced fractions $\frac{p}{q}$. If p and q are not relatively prime, a better approximation is achieved as a smaller p can be obtained by the reduction of the fraction.

Bresen.java **Page 1/3**

```
 1: import  java.io.PrintWriter;
 2: import  java.io.IOException;
 3:
 4: import  java.util.ArrayList;
 5: import  java.util.Collection;
 6: import  java.util.HashMap;
 7: import  java.util.HashSet;
 8: import  java.util.Iterator;
 9: import  java.util.List;
10:
11: public class Bresen {
12:     /**
13:      * Structure to hold the parameters of a cycle and providing
14:      * a method for its evaluation.
15:      */
16:     private static class Cycle {
17:       public final int       k;
18:       public final Fraction  qu;
19:
20:       public Cycle(int k, Fraction qu) {
21:         this.k  = k;
22:         this.qu = qu;
23:       }
24:     }
25:
26:     /**
27:      * Implements the quality measure, here the mean quadratic error to
28:      * the perfect edges of the output clock normalized to (0,1] by
29:      * $\frac{1}{1+\sqrt{x}}$.
30:      */
31:     private static Fraction eval(int p, int q, List<Integer> log) {
32:       final int      s  = log.size();         // toggle event count
33:       final Fraction pp = new Fraction(q, p); // perfect period
34:       Fraction       ph = new Fraction(0, 1); // phase to be determined
35:
36:       Iterator<Integer>  it;
37:       for(it = log.iterator(); it.hasNext(); ph.add(it.next()));
38:       ph.div(s);
39:       ph.sub(new Fraction(pp).div(2).mul(s-1));
40:
41:       Fraction  sd2 = new Fraction(0, 1); // sum of distance (error) squares
42:       it = log.iterator();
43:       for(int i = 0; it.hasNext();) {
44:         sd2.add(new Fraction(-i++, 1).mul(pp).sub(ph).add(it.next()).sqr());
45:       }
46:       return  sd2.div(s);
47:     }
48:
49:     /**
50:      * Determines all cycles for the quotient $\frac{p}{q}$ in the
51:      * ModModel m.
52:      */
53:     private static Collection<Cycle> sim(int p, int q, ModModel m) {
54:       /** Result Collection of all Cycles as List. */
55:       final ArrayList<Cycle>  res = new ArrayList<Cycle>();
56:
57:       /** Set of representatives of start value mod class whose
58:        *  cycle they eventually lead into is already known. */
59:       final HashSet<CS>  solved = new HashSet<CS>();
60:
61:       /** Prefetch a few model-specific values into local variables. */
62:       final int  n   = m.n(p, q);  // bit width n
63:       final int  t   = m.t();      // threshold t in {0,1}
64:       final int  msh = n-1;        // MSB-Shift
65:       final int  msk = ~(~0<<n);   // Maske
66:
67:       /** Set of representatives of start value mod class already
68:        *  seen during the simulation for the current start value.
69:        *  All of these eventually lead into the same cycle.
70:        *  They are mapped to the count of blocks of q simulation
71:        *  steps to help determine the period as k*q. */
72:       final HashMap<CS, Integer>  seen = new HashMap<CS, Integer>();
```

Bresen.java **Page 2/3**

```
 73:
 74:       /** Discrete time points of toggle events as the number of the
 75:        *  corresponding input clock edge. Used for quality evaluation. */
 76:       final ArrayList<Integer>  log  = new ArrayList<Integer>();
 77:
 78:       for(final CS e : m.startValues(p, q)) {
 79:         // only simulate for representatives not already encountered
 80:         if(!solved.contains(e)) {
 81:           seen.clear();
 82:           log .clear();
 83:           seen.put(e, 0);
 84:
 85:           int  es = e.s();  // pseudo-components of e-register
 86:           int  ec = e.c();
 87:           int  i  = 0;      // counter for finished q-cycles
 88:           sim: while(true) {
 89:             // simulate a full q-cycle
 90:             for(int j = 0; j < q; j++) {
 91:               final boolean  mod = (es>>msh)+(ec>>msh) > t;
 92:               final int  d   = mod? p-q : p;
 93:               final int  esp = msk & (es^ec^d);
 94:               final int  ecp = msk & (((es&ec)|(es&d)|(ec&d)) << 1);
 95:
 96:               // drop out if we had an addition mod 2^n
 97:               if((es+ec+p-esp-ecp)%q != 0)  break  sim;
 98:               es = esp;
 99:               ec = ecp;
100:               if(mod)  log.add(i*q + j);
101:             }
102:
103:             final CS  ee = new CS(es, ec);
104:
105:             // reached a representative already solved?
106:             if(solved.contains(ee))  break  sim;
107:
108:             // closed a cycle?
109:             final Integer  i0 = seen.put(ee, ++i);
110:             if(i0 != null) {
111:               final int  k = i-i0; // number of q-cycles (period is k*q)
112:
113:               // evaluate cycle and add it to solution
114:               // the last k*p log-entries matter
115:               final int  s = log.size();
116:               res.add(new Cycle(k, eval(p, q, log.subList(s-k*p, s))));
117:               break  sim;
118:             }
119:           }
120:           // mark all representatives seen in this simulation as solved
121:           solved.addAll(seen.keySet());
122:         }
123:       }
124:       return  res;
125:     }
126:
127:     /**
128:      * EUKLIDian algorithm to determine the greatest common factor (gcf).
129:      */
130:     private static int gcf(int a, int b) {
131:       while(b != 0) {
132:         final int  bb = b;
133:         b = a%bb;
134:         a = bb;
135:       }
136:       return  a;
137:     }
138:
139:     /**
140:      * Determine cycles of all quotients with p<q and L<=q<=U for all available
141:      * ModModels and output results to dat-file "bresen<L>_<U>.dat".
142:      */
143:     private static void simRange(final int L, final int U) {
144:       PrintWriter  out     = null;
```

```
145:       Thread[]     threads = new Thread[ModModel.models.size()];
146:
147:       try {
148:         final Collector clct =
149:           new Collector(out = new PrintWriter("bresen"+L+"_"+U+".dat"), 1000);
150:
151:         // simulation for ModModels
152:         for(final ModModel  m : ModModel.models) {
153:           final Thread  t = new Thread() {
154:             public void run() {
155:               for(int q = L; q <= U; q++) {
156:                 for(int p = 1; p < q; p++) {
157:                   if(gcf(q, p) > 1)  continue;
158:
159:                   // simulate model for these p and q
160:                   final Collection<Cycle>  cycles = sim(p, q, m);
161:
162:                   // obtain cycle length distribution and best quality
163:                   Cycle  best = null;
164:                   for(final Cycle cyc : cycles) {
165:                     if((best == null)||(best.qu.compareTo(cyc.qu) > 0))
166:                       best = cyc;
167:                   }
168:
169:                   clct.addEntry(m, p, q,
170:                         new Collector.Entry(best.qu, best.k, cycles.size()));
171:                 }
172:               }
173:             }
174:           };
175:           t.start();
176:           threads[m.getIndex()] = t;
177:         }
178:
179:         // wait for threads to finish
180:         for(final Thread  t : threads) {
181:           try { t.join(); } catch(InterruptedException e) {}
182:         }
183:       }
184:       catch(IOException e) {
185:         e.printStackTrace();
186:       }
187:       finally {
188:         if(out != null)  out.close();
189:       }
190:     }
191:
192:     public static void main(String[] args) throws Exception {
193:         simRange(Integer.parseInt(args[0]), Integer.parseInt(args[1]));
194:     }
195: }
```



```
 1: /**
 2:  * Represents a number in carry-save format. Equality is established
 3:  * without regard of the coding of a 1-valued digit position.
 4:  * A strict order consistent with this notion of equality is
 5:  * established. Hash codes of equal carry-save representations
 6:  * equal as well.
 7:  */
 8: public class CS implements Comparable<CS> {
 9:     private final int  v;  // numeric value
10:     private final int  x;  // 1-valued digits (XOR of pseudo-components)
11:
12:     public CS(int s, int c) {
13:         this.v = s+c;
14:         this.x = s^c;
15:     }
16:
17:     public int s() { return  x | ((v-x)>>1); } // digits 1 and 2
18:     public int c() { return      ((v-x)>>1); } // only digit 2
19:
20:     public int hashCode() { return  v^x; }
21:
22:     public boolean equals(Object o) {
23:         if(o instanceof CS) {
24:             CS  y = (CS)o;
25:             return  (v == y.v) && (x == y.x);
26:         }
27:         else  return  false;
28:     }
29:     public int compareTo(CS o) {
30:         int  d = v - o.v;
31:         return (d != 0)? d : x - o.x;
32:     }
33:
34:     public String toString() {
35:         final StringBuilder  buf = new StringBuilder();
36:         for(int ss = s(), xx = x; ss > 0; ss>>=1, xx>>=1) {
37:             buf.append(((ss&1) == 0)? '0' : ((xx&1) == 0)? '2' : '1');
38:         }
39:         return  buf.reverse().toString();
40:     }
41: }
```

ModModel.java **Page 1/2**

```
 1: import  java.util.ArrayList;
 2: import  java.util.HashSet;
 3: import  java.util.Set;
 4:
 5: /**
 6:  * Represents a model determining the occurence of modulo events.
 7:  * This abstract class implements a generic method calculating
 8:  * a set of start values consisting of all carry-representations
 9:  * of all representatives of the module class of $2^n-1$ within
10:  * the cyclic range E. Concrete subclasses must specify the
11:  * implementation bit width in dependence on q, fix the threshold
12:  * to one of the values 0 or 1 and provide a name.
13:  */
14: public abstract class ModModel {
15:     private int idx = 0;
16:     private void setIndex(int idx) { this.idx = idx; }
17:     public  int  getIndex() { return  idx; }
18:
19:     abstract public String name();
20:     abstract public int n(int p, int q);
21:     abstract public int t();
22:
23:     Set<CS> startValues(int p, int q) {
24:         final HashSet<CS>  res = new HashSet<CS>();
25:         final int          n   = n(p, q);
26:         final int          s0  = (1<<n)-1;
27:
28:         // uniquely-coded representative $2^n-1$ of the mod class
29:         res.add(new CS(s0, 0));
30:
31:         int  kl, ku;  // lower and upper bound of k != 0
32:         switch(t()) {
33:         case 0:
34:             // there may be smaller valid representatives of same class
35:             kl = (p+1-(1<<(n-1)))/q - 1;
36:             ku = -1;
37:             break;
38:
39:         case 1:
40:             // there may be larger valid representatives of same class
41:             kl = 1;
42:             ku = ((1<<(n-1))-1+p)/q;
43:             break;
44:
45:         default:
46:             throw new IllegalArgumentException("Threshold must be 0 or 1");
47:         }
48:
49:         // add all carry-save representations of all other valid
50:         // representatives of mod class
51:         for(int k = kl; k <= ku; k++) {
52:             final int  i  = s0 + k*q;
53:             final int  js = 2-(i&1);          // step
54:             final int  jl = -js & (i+2)/2;  // lower bound
55:             final int  ju = (i<s0)? i : s0; // upper bound
56:
57:             // sliding additive partition of value i
58:             for(int j = jl; j <= ju; j += js) {
59:                 // equivalent representations are automatically discarded
60:                 res.add(new CS(j, i-j));
61:             }
62:         }
63:
64:         return  res;
65:     }
66:
67:     protected static int ldceil(int x) {
68:         int  r = 0;
69:
70:         x--;
71:         while(x > 0) {
72:             r++;
```

ModModel.java **Page 2/2**

```
 73:             x>>=1;
 74:         }
 75:         return  r;
 76:     }
 77:
 78:     /**
 79:      * Concrete implementations of ModModel.
 80:      */
 81:     public static final ArrayList<ModModel> models = new ArrayList<ModModel>();
 82:     private static void addModel(ModModel m) {
 83:         m.setIndex(models.size());
 84:         models.add(m);
 85:     }
 86:
 87:     static {
 88:       addModel(new ModModel() {
 89:               public String name() { return "T0"; }
 90:               public int n(int p, int q) {
 91:                   final int  na =     ldceil(p+1);
 92:                   final int  nb = 1 + ldceil(q-p);
 93:                   return (na > nb)? na : nb;
 94:               }
 95:               public int t() { return  0; }
 96:           });
 97:       addModel(new ModModel() {
 98:               public String name() { return "T1"; }
 99:               public int n(int p, int q) {
100:                   final int  na = p+1;
101:                   final int  nb = q-p;
102:                   return  1 + ldceil((na > nb)? na : nb);
103:               }
104:               public int t() { return  1; }
105:           });
106:       addModel(new ModModel() {
107:               public String name() { return "U0"; }
108:               public int n(int p, int q) {
109:                   final int  na =     ldceil(p+1);
110:                   final int  nb = 1 + ldceil(q-p);
111:                   return  1 + ((na > nb)? na : nb);
112:               }
113:               public int t() { return  0; }
114:           });
115:       addModel(new ModModel() {
116:               public String name() { return "U1"; }
117:               public int n(int p, int q) {
118:                   final int  na = p+1;
119:                   final int  nb = q-p;
120:                   return  2 + ldceil((na > nb)? na : nb);
121:               }
122:               public int t() { return  1; }
123:           });
124:     }
125: }
```

Collector.java **Page 1/2**

```
  1: import  java.io.PrintWriter;
  2:
  3: import  java.util.HashMap;
  4:
  5: public class Collector {
  6:     public static class Entry {
  7:         public final Fraction  quality;
  8:         public final int       period;
  9:         public final int       cycles;
 10:
 11:         public Entry(Fraction quality, int period, int cycles) {
 12:             this.quality = quality;
 13:             this.period  = period;
 14:             this.cycles  = cycles;
 15:         }
 16:     }
 17:
 18:     private static class Key {
 19:         public final int  p;
 20:         public final int  q;
 21:
 22:         public Key(int p, int q) {
 23:             this.p = p;
 24:             this.q = q;
 25:         }
 26:
 27:         public boolean equals(Object o) {
 28:             if(o instanceof Key) {
 29:                 Key  k = (Key)o;
 30:                 return (p == k.p)&&(q == k.q);
 31:             }
 32:             else  return  false;
 33:         }
 34:
 35:         public int hashCode() {
 36:             return (((q-2)*(q-1))>>1) + p;
 37:         }
 38:     }
 39:
 40:     private static class Line {
 41:         private final Key  k;
 42:
 43:         private Entry[]  entries;
 44:         private int      empty;
 45:
 46:         public Line(Key k) {
 47:             this.k = k;
 48:             this.entries = new Entry[this.empty = ModModel.models.size()];
 49:         }
 50:
 51:         public boolean addEntry(ModModel m, Entry e) {
 52:             entries[m.getIndex()] = e;
 53:             return  --empty == 0;
 54:         }
 55:
 56:         public void print(PrintWriter out) {
 57:           {
 58:             Fraction  bq  = new Fraction(1, 0);
 59:             int       msk = 0;
 60:
 61:             for(final Entry e : entries) {
 62:               if(e.quality.compareTo(bq) < 0)  bq = e.quality;
 63:             }
 64:             for(int i = 0; i < entries.length; i++) {
 65:               if(entries[i].quality.equals(bq))  msk |= 1 << i;
 66:             }
 67:
 68:             out.printf("%4d %4d %4d\t", k.p, k.q, msk);
 69:           }
 70:           for(int i = 0; i < entries.length; i++) {
 71:             final Entry  e = entries[i];
 72:             out.printf("%8d/%8d %2d %2d\t", e.quality.a(), e.quality.b(),
```

Collector.java **Page 2/2**

```
 73:                                            e.period, e.cycles);
 74:           }
 75:           out.println();
 76:         }
 77:     }
 78:
 79:     private final int  BUF_CAP;
 80:     private final HashMap<Key, Line>  lines;
 81:     private final PrintWriter         out;
 82:
 83:     public Collector(PrintWriter out, int bufCap) {
 84:         this.BUF_CAP = bufCap;
 85:         this.lines   = new HashMap<Key, Line>();
 86:         this.out     = out;
 87:
 88:         final StringBuilder  bld = new StringBuilder("# Modells:");
 89:         final int            n   = ModModel.models.size();
 90:         for(int i = 0; i < n;) {
 91:             bld.append('\t').append(ModModel.models.get(i++).name());
 92:         }
 93:         out.println(bld);
 94:     }
 95:
 96:     private Line getLine(Key k) {
 97:       while(true) {
 98:         Line l = lines.get(k);
 99:         if(l != null)  return  l;
100:         if(lines.size() < BUF_CAP) {
101:           lines.put(k, l = new Line(k));
102:           return  l;
103:         }
104:         try { wait(); } catch(InterruptedException ex) {}
105:       }
106:     }
107:
108:     public synchronized void addEntry(ModModel m, int p, int q, Entry e) {
109:         final Key   k = new Key(p, q);
110:         final Line  l = getLine(k);
111:         if(l.addEntry(m, e)) {
112:             l.print(out);
113:             lines.remove(k);
114:             notifyAll();
115:         }
116:     }
117: }
```

Fraction.java Page 1/2

```
  1: public class Fraction implements Comparable<Fraction> {
  2:     private long  a;
  3:     private long  b;
  4:
  5:     public Fraction(int a, int b) {
  6:         this.a = a;
  7:         this.b = b;
  8:         reduce();
  9:     }
 10:     public Fraction(int a)      { this(a, 1);     }
 11:     public Fraction(Fraction o) {
 12:         a = o.a;
 13:         b = o.b;
 14:     }
 15:
 16:
 17:     public long a() { return  a; }
 18:     public long b() { return  b; }
 19:
 20:
 21:     public int compareTo(Fraction o) {
 22:         final long  d = a*o.b - o.a*b;
 23:         return (d < 0)? -1 : (d > 0)? 1 : 0;
 24:     }
 25:     public boolean equals(Object o) {
 26:         if(o instanceof Fraction) {
 27:             final Fraction  f = (Fraction)o;
 28:             return (a == f.a) && (b == f.b);
 29:         }
 30:         return  false;
 31:     }
 32:
 33:
 34:     private void reduce() {
 35:         long  aa = a;
 36:         long  bb = b;
 37:
 38:         while(bb != 0) {
 39:             final long  bbb = bb;
 40:             bb = aa % bb;
 41:             aa = bbb;
 42:         }
 43:         if((a < 0) || (b < 0)) {
 44:             if(b < 0)  aa = -Math.abs(aa);
 45:             else       aa =  Math.abs(aa);
 46:         }
 47:
 48:         a /= aa;
 49:         b /= aa;
 50:     }
 51:
 52:     public Fraction add(int o) {
 53:         a += b*o;
 54:         return  this;
 55:     }
 56:     public Fraction add(Fraction o) {
 57:         a  = a*o.b + b*o.a;
 58:         b *= o.b;
 59:         reduce();
 60:         return  this;
 61:     }
 62:
 63:     public Fraction sub(int o) {
 64:         a -= b*o;
 65:         return  this;
 66:     }
 67:     public Fraction sub(Fraction o) {
 68:         a  = a*o.b - b*o.a;
 69:         b *= o.b;
 70:         reduce();
 71:         return  this;
 72:     }
```

Fraction.java Page 2/2

```
 73:
 74:     public Fraction mul(int o) {
 75:         a *= o;
 76:         reduce();
 77:         return  this;
 78:     }
 79:     public Fraction mul(Fraction o) {
 80:         a *= o.a;
 81:         b *= o.b;
 82:         reduce();
 83:         return  this;
 84:     }
 85:
 86:     public Fraction div(int o) {
 87:         b *= o;
 88:         reduce();
 89:         return  this;
 90:     }
 91:     public Fraction div(Fraction o) {
 92:         a *= o.b;
 93:         b *= o.a;
 94:         reduce();
 95:         return  this;
 96:     }
 97:
 98:     public Fraction sqr() {
 99:         a *= a;
100:         b *= b;
101:         return  this;
102:     }
103:
104:     public String toString() {
105:         return  new StringBuilder().append('(').append(a).append('/')
106:                                    .append(b).append(')').toString();
107:     }
108: }
```