Abstract—This paper presents a robust, low-cost ADC code hit counting technique to record the number of times each ADC output code word appears with respect to the ramp input. Using a smart center code tracking engine, the proposed code hit counter performs robustly against the code transition noise, missing code segments, and non-monotonicity; furthermore, the required hardware and test time is at the same level as the known best results. The robustness together with the low overhead makes the proposed code hit counter suitable for (on-line) ADC self-testing and self-calibration applications.

I. INTRODUCTION

The analog-to-digital converter (ADC) is a widely used mixed-signal component in modern System-on-Chip (SoC) designs. Testing embedded ADCs, however, is costly and time consuming because it is mainly specification based functional testing [1]. To lower the ADC test cost, many design-for-test (DFT) and built-in-self-test (BIST) techniques have been developed [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. Most of them aim to reduce the ATE (automatic test equipment) performance requirement by realizing on-chip test signal generation or test response analysis.

ADCs are tested for static or dynamic performance parameters. This work relates to static ADC testing which concerns the offset, gain, DNL (differential non-linearity), and INL (integral non-linearity). Deriving these static parameters requires knowledge of the code transition edges; however, direct code edge measurement is complicated and time consuming, especially for high-resolution ADCs. The common practice is to determine the code edge positions from the code widths which, in turn, are derived from the ADC code hits with respect to a test input with known amplitude distribution [15]. (The code hit of an ADC output code is the number of times the code appears.)

The linear ramp is frequently adopted for code width derivation because it incurs the least post-processing efforts—the code width is proportional to the code hit. Figure 1 depicts a ramp-based ADC BIST architecture; the test related circuits include the ramp generator, the code hit counter, and the code hit analyzer. In [16], [17], the authors proposed adaptive ramp generation techniques that allow precise slope control. Code hit counting and the following code hit analysis techniques were investigated in [12], [13], [14], [18].

This work concerns the code counting technique. Figure 2 shows an ADC output sequence with respect to a rising ramp input. This sequence exhibits missing codes (code 13), non-monotonicity (the transition from code 15 to 8), and code transition noise (the shaded regions) which, if ignored, may jeopardize the code hit counting operations. A practical code hit counter should perform robustly against these non-idealities while incurring as little silicon and test time overhead as possible.

Most external testers use the histogram technique to record the code hits with respect to the linear ramp input. The limitation is the associated exponential memory requirement (with respect to the ADC resolution), which is not always available on chip. The time decomposition technique in [9], [10] employs time-spread instead of concurrent code hit counting; this minimizes the hardware overhead but adversely adds an exponential complexity factor to the test time. In [12], [13], [14], the authors proposed compact code hit counting circuits that achieve the same level of test time as the histogram approach and hardware overhead as the time decomposition approach; however, code transition noise is not considered. In [18], the authors proposed to concurrently count the hits of multiple codes around the center code; however, it is not discussed how to update the codes and how to handle the missing code segments and non-monotonicity.

This paper presents a robust, low-cost code hit counter for...
(on-line) ADC self-testing and self-calibration applications. It is able to output the code hits for practical ADC output sequences like that in Figure 2. The enabling technique of the code hit counter is a smart center code tracking engine. It keeps a list of monitored codes (around the center code) and the code hit counter is a smart center code tracking engine. It sequences like that in Figure 2. The enabling technique of (on-line) ADC self-testing and self-calibration applications.

A brief comparison of the proposed technique with prior arts is shown in Table I. To our best knowledge, it is the first code hit counting technique that performs robustly in the presence of code transition noise, missing code segments, and non-monotonicity without adding an exponential complexity factor to the hardware or test time overhead.

This paper is organized as follows. In Section II, we briefly introduce the linear ramp based ADC testing and the code transition noise modeling. Then, Section III and Section IV describe the basic and complete code hit counting techniques, respectively. Simulation results are described in Section V and we conclude this work in Section VI.

II. PRELIMINARIES

A. Linear histogram testing

Linear histogram testing is a popular static ADC testing technique; it applies a linear ramp to the ADC under test and records the code hits. The collected code hits form the code histogram. Because the input is linear, the code width is proportional to the code hit.

Consider an $n$-bit ADC and let $H(i)$ be the code hit associated with code $i$. The average code hit $H_{\text{avg}}$ is as follows.

$$H_{\text{avg}} = \frac{\sum_{i=1}^{2^n-2} H(i)}{2^n-2}$$

In (1), $H(0)$ and $H(2^n - 1)$ are excluded because they are singly bounded. Choice of $H_{\text{avg}}$ is a tradeoff between test time and test resolution—the former is proportional while the latter is inversely proportional to $H_{\text{avg}}$. In general, $H_{\text{avg}}$ is between 16 and 32.

Assuming that the reference line is the end-point line which connects the first and last transition edges, the ADC DNL and INL can be derived as follows.

$$DNL(i) = \frac{H(i) - H_{\text{avg}}}{H_{\text{avg}}} \text{ LSB}$$ (2)

$$INL(i) = \sum_{j=1}^{i} DNL(j)$$ (3)

(2) and (3) suggest that, the code hit analyzer can compute DNL and INL in an on-the-fly manner without having to store all the code hits if it receives the code hits in a monotonic order. Furthermore, one can realize the division with shift operations if $H_{\text{avg}}$ is made a power of two (through ramp slope control).

B. The maximum code jump $J$

Because of circuit noise, the ADC output codes exhibit a distribution even when the input to the ADC is a DC value. When one applies a ramp to the ADC input, this results in non-zero code transition regions (the shaded regions in Figure 2). Instead of jumping directly to the next code word when reaching the code edge, the ADC output settles down to the next code word after a noise-dependent transition region [19].

In the following discussion, we define the maximum code jump $J$ as follows.

Definition 1: Given an ADC output sequence $Q$ with respect to a test input $I$, the maximum code jump $J$ associated with $Q$ is the maximum deviation of $Q$ from the transition noise free ADC output sequence $Q_{\text{ideal}}$ with respect to the same test input $I$.

The amount of code transition noise affects the code hit counter implementation. In the proposed technique, the user specified maximum code jump determines the amount of registers and counters necessary to store the monitored codes and code hits.

Given a real ADC output sequence, it is impossible to determine $J$ because the ideal sequence is unavailable for comparison. Furthermore, the noise may be so small that it only delays or brings forward the code transition by one cycle without causing any visible code transition region. For example, consider the two sequences below. They differ in only one output (the underlined one); there is no telling which is the ideal one.

$$\cdots 444444555555666666 \cdots$$

$$\cdots 444444555555666666 \cdots$$

III. THE BASIC CODE HIT COUNTING TECHNIQUE

We first introduce the basic code hit counting technique assuming that the ADC is monotonic and has no missing code.

<table>
<thead>
<tr>
<th>memory* (# words)</th>
<th>time decomposition [9], [10]</th>
<th>code width [12], [13], [14]</th>
<th>[18]</th>
<th>proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>test time (# ramps)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>code transition noise tolerance</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>missing code segment tolerance</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>non-monotonicity tolerance</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

*: the number of registers needed for code hit counting
A. The basic code hit counter architecture

For the proposed code hit counter, the code that it regards as the most probable to appear next is called the center code, denoted by $CC$. The code hit counter uses a counter to store the center code. To tolerate the noise-induced code deviation, it monitors a list of $3J + 1$ codes concurrently. These codes are continuous and spread around $CC$. For example, given $J = 2$, if $CC = 10$, the list of monitored codes is: $6, 7, 8, 9, 10, 11, 12$. Note that the actual center code may be $8, 9, or 10$—the ambiguity is due to the code transition noise. Since the monitored codes are continuous, only the center code is explicitly stored (in the center code counter). However, the code hit counter needs $3J + 1$ hit counters to store the code hits associated with the monitored codes. Upon receiving a new code from the ADC, the code hit counter compares it with the monitored code list and performs one or both of the following tasks:

- Increase the corresponding hit counter by one.
- Increase $CC$ by one and shift the contents of the hit counters.

Figure 3 depicts the basic code hit counter architecture; it consists of a center code counter, a code tracking engine, and $3J + 1$ hit counters. The center code counter is a basic counter—it is reset to zero at the beginning and increased by one when the code tracking engine decides to increase the center code to adapt to the new ADC output code. The $3J + 1$ two-input counters (TICs) store the hits of the monitored codes; the $i$th hit counter $M_i$ stores the hit code of code $CC + i - (2J+1)$.

The truth table of the TIC is depicted in Table II. If $inc$ equals 1, TIC increases its value by one; otherwise, it retains its value if $ld$ equals zero or load from one of $D1$ and $D2$ depending on $sel$. Using the two-input counters allows the code hit counter to shift the hit counter values and increase the value of one of them in one clock cycle.

Finally, whenever the $SH$ signal is raised, a new code hit is ready. The code hit analyzer then loads the code and its code hit for further analysis. The $ERR$ signal becomes one for one cycle if the input code fails the maximum jump code constraint due to excessive noise, missing codes, or non-monotonicity.

![Fig. 3. The robust code hit counter (basic version).](image)

<table>
<thead>
<tr>
<th>$inc$</th>
<th>$ld$</th>
<th>$sel$</th>
<th>$Q^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$D1$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$D2$</td>
</tr>
</tbody>
</table>

$Q^+$: next state

### Table III
<p>| Truth Table of the Two-Input Counter (TIC) |
|-----------------|----------------|</p>
<table>
<thead>
<tr>
<th>$inc$</th>
<th>$ld$</th>
<th>$sel$</th>
<th>$Q^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$D1$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$D2$</td>
</tr>
</tbody>
</table>

$Q^+$: next state

B. The basic code tracking algorithm

Before describing the basic code tracking algorithm, we define in Table III the four possible input conditions: $OR^-$, $LH$, $UH$, and $OR^+$ according to the ADC output code $x$, the current center code $CC$, and the maximum code jump $J$. Since the list of monitored codes is monotonic and continuous, one and exactly one of the four conditions is true.

The basic code tracking rules are listed in Table IV. The first column corresponds to the four input conditions. The second column lists the $SH$ signal. Column three shows the values of the $sel$ inputs to the hit counters. In this column, a bold I means $sel_i = 0$ for $i = 1, 2, \cdots, 3J + 1$; otherwise, the one and only one non-zero $sel$ signal is shown. Column four lists the values of the $inc$ inputs. The same notation as column three applies. The last column is the $ERR$ signal.

Let $x$ be the ADC input code. The tracking rules are detailed below.

- **$OR^-$**: $x$ is out of the tracker’s tracking range; the reason is excessive code transition noise or non-monotonicity. The $ERR$ signal is raised to signal the erroneous input condition. The code hit counter does nothing and skips this code.
- **$LH$**: $x$ matches one code in the lower half of the monitored list or the center code. The hit counter that corresponds to $x$, i.e., $M_{x-CC+2J+1}$, is increased by one by setting its $inc$ input to high.
- **$UH$**: $x$ matches one code in the upper half of the monitored code list. The code tracking algorithm increases $CC$ by one (by setting $SH = 1$) because $x$ has exceeded $CC$. For the hit counters, we have to update the hit

### Table IV
<table>
<thead>
<tr>
<th>Basic Tracking Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>input condition</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>$OR^-$</td>
</tr>
<tr>
<td>$LH$</td>
</tr>
<tr>
<td>$UH$</td>
</tr>
<tr>
<td>$OR^+$</td>
</tr>
</tbody>
</table>
count of code \( x \) and shift the hit counter values in one cycle. We note that the updated hit of \( x \) will be stored in \( M_{x-CC+2J} \) after the shift; thus, we achieve the two tasks in one cycle by shifting the contents of the hit counters (\( SH = 1 \)) with the exception that \( M_{x-CC+2J} \) loads its input from the \( Q+1 \) output of \( M_{x-CC+2J+1} \) (by setting \( sel_{x-CC+2J} = 1 \)).

- **OR**: \( x \) is out of the tracking range; the reason is excessive code transition noise or a missing code segment. The \( ERR \) signal is raised to signal the erroneous input condition. Because \( x \) does not match any monitored code, it is skipped. However, \( SH \) is set to one to increase \( CC \) and shift the hit counter contents; the reason is for \( CC \) to catch up with the ADC output sequence.

An example with \( J = 2 \) is given in Figure 4 to illustrate the basic code tracking algorithm. At the beginning, the center code is \( CC = 12 \) which corresponds to \( R_7 \). Since \( J = 2 \), the list of monitored codes is \( 8, 9, 10, 11, 12, 13, 14 \). In cycles 4, 7, 8, 9, and 13, the input condition is \( UH \). The center code \( R_5 \) is increased by one; the hit counter values are shifted and the one that corresponds to the input code is increased. New code hit results are available for the following code hit analyzer. In the other cycles, the input condition is \( LH \); the corresponding hit counter is increased. Although not designed to do so, the basic code hit counter works correctly in the existence of the missing code 14.

**C. Discussions**

1) **Area overhead**: It is interesting that the required number of counters depends on the maximum code jump \( J \), but not on the ADC resolution. Of course, the center code counter and the arithmetic circuits have to be wider for higher resolution ADCs. Note that \( M_{3J+1} \) is redundant; one can remove it and replace its “\( Q \)” and “\( Q+1 \)” outputs with 0 and 1, respectively. That the \( D1 \) and \( D2 \) inputs of \( M_{3J} \) is fixed also allows further optimization of \( M_{3J} \). Finally, \( M_{2J+1}, M_{2J+2}, \ldots, M_{3J+1} \) can be further reduced because their \( inc \) inputs are always zero.

2) **Tolerance to missing codes and non-monotonicity**: Although not designed to handle missing codes and non-monotonicity, the basic code tracking technique has limited capability of tolerating missing code segments and non-monotonicity to up to \( J \) code deviations. This has been shown in the example in Figure 4.

**IV. THE COMPLETE CODE HIT COUNTING TECHNIQUE**

In this section, we introduce the complete code hit counting technique that is designed to handle missing code segments and non-monotonicity.

A. **The complete code hit counter algorithm**

Although the basic code hit counter has limited capability of handling missing codes and non-monotonicity, it is impractical for cases with unknown or severe missing codes and non-monotonicity. The complete code hit counter is developed to explicitly handle arbitrary missing code segments and non-monotonicity.

To properly handle the missing codes and non-monotonicity, the complete code hit tracking technique differs from the basic one in the following aspects.

- To maintain the property that the area overhead depends on the maximum code jump, not the severity of missing code segments and non-monotonicity, the list of monitored codes may be discontinuous or non-monotonic.
- Because the monitored list may be discontinuous, the code hit counter has to explicitly store all the monitored codes, instead of just the center code as in the basic code hit counter.
- It is harder to predict the center code; thus, the complete code hit tracker has to monitor more codes.
- The code hit counter needs to know whether the list of monitored codes is continuous and monotonic or not.

The complete code hit counter monitors \( 6J - 1 \) codes concurrently. These codes and their corresponding code hits are stored in the \( 6J - 1 \) code registers \((R_1, R_2, \ldots, R_{6J-1})\) and \( 6J - 1 \) hit counters \((M_1, M_2, \ldots, M_{6J-1})\), respectively. Figure 5 depicts the complete code tracking algorithm which consists of three modes.

- **Continuous mode** (\( DC = 0 \)). In this mode, the code registers are continuous. The controller utilizes a disconti-

---

**Table 1**

<table>
<thead>
<tr>
<th>( x )</th>
<th>( R_1 )</th>
<th>( R_2 )</th>
<th>( R_3 )</th>
<th>( R_4 )</th>
<th>( R_5 )</th>
<th>( R_6 )</th>
<th>( R_7 )</th>
<th>( M_1 )</th>
<th>( M_2 )</th>
<th>( M_3 )</th>
<th>( M_4 )</th>
<th>( M_5 )</th>
<th>( M_6 )</th>
<th>( M_7 )</th>
<th>( code )</th>
<th>( hit )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>13</td>
<td>17</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>12</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>
nuity counter to keep track of the code register continuity. The code registers are discontinuous if $DC \neq 0$.

- **Missing code mode** ($MC = 1$). In this mode, the code registers are discontinuous but monotonic, which is due to missing code segments. When in the continuous mode, if the controller detects a missing code segment, it sets $MC$ to 1 and $DC$ to $4J - 1$; this brings the code hit counter to the missing code mode.

- **Non-monotonic mode** ($NM = 1$). In this mode, the code registers are non-monotonic, which is due to ADC output non-monotonicity. When in the continuous mode, if the controller detects non-monotonicity, it sets $NM$ to 1 and $DC$ to $4J - 1$; this brings the code hit counter to the non-monotonic mode.

Note that (1) one and exactly one of $DC = 0$, $MC = 1$, and $NM = 1$ is true, and (2) the code hit counter may malfunction if it encounters missing codes or non-monotonicity when not in the continuous mode. The operations in Figure 5 are explained below.

- **split**. This is performed when missing codes or non-monotonicity is detected in the continuous mode. Let $x$ be the input code. $R_1$ to $R_{4J-1}$ remain the same; $R_{4J}$ to $R_{6J-1}$ are updated as follows.

  $R_k^+ = k + x - 5J + 1$  \hspace{1cm} (4)

where $R_k^+$ denotes the next state of $R_k$.

- **shift**. The contents of the code registers are shifted according to

  $R_{j \neq 6J-1}^+ = R_{j+1}$ \hspace{1cm} (5)
  $R_{6J-1}^+ = R_{6J-1} + 1$ \hspace{1cm} (6)

The count registers are updated the same way except that $M_{6J-1} = 0$.

- **add**. The hit counter that corresponds to the input code $x$ is increased by one.

- **$DC$--**. This operation decreases the $DC$ value by one. If $DC$ becomes zero, either $MC$ or $NM$ is reset to 0.

An example with $J = 1$ is given in Figure 6 to illustrate the complete code tracking algorithm. From cycles 1 to 6, the code hit counter is in the continuous mode; its behavior is similar to that of the basic code hit counter. In cycle 7, the code hit counter enters the missing code mode due to the input code $x = 18$ in cycle 6. One can see from the the last three columns that $DC$ becomes $4J - 1 = 3$ and $MC$ is set to 1. In cycle 13, the code registers become continuous and monotone again. In cycle 17, the code hit counter enters the non-monotonic mode due to $x = 7$ in cycle 16. All the code hits are correctly reported.

V. SIMULATION RESULTS

Behavior simulations are performed to validate the proposed code hit counting techniques.

A. Basic code hit counting

We generate 1,000 ADC output sequences as follows.

1) Generate a sequence of $0 \cdots 2^n - 1$ where $n$ is the ADC resolution.

2) Expand the sequence by assigning code widths. Code widths are randomly perturbed from the average width but remain positive, i.e., no missing code.

3) Each code $x$ in the sequence is randomly perturbed by at most $J$.

Note that the third step may cause the generated sequences to contain missing codes. For example, $\cdots 55667788 \cdots$, after perturbation, could become $\cdots 55558888 \cdots$, in which codes 6 and 7 are missing.
The generated sequences are applied to the basic code hit counter. Out of the 1,000 sequences, the basic counter reports errors for 3 sequences (due to missing codes). For the other sequences, the basic counter correctly outputs the code hits.

### B. Complete code hit counting

1,000 sequences are generated using the procedure for the basic counter except that the sequence in step one may contain missing codes or non-monotonicity. Out of the 1,000 sequences, the complete counter reports error for 1 sequence (due to close missing code segments). For the other sequences, the basic counter outputs correct code hits for all but one sequence.

### VI. Conclusion

This work presents a low-cost code hit counting technique which performs robustly in the presence of code transition noise, missing code segments, and non-monotonicity. Behavior-level simulation results validate the robustness of the proposed technique. In the future, we will (1) perform thorough verification on the algorithm, (2) further optimize the hardware, and (2) seek hardware validation.

### References


