APPLICATION OF SKIPSM TO BINARY CORRELATION

Frederick M. Waltz
2095 Delaware Avenue, Mendota Heights, MN 55118-4801 USA

ABSTRACT

Binary correlation is often used for finding specified patterns in complex binary images, especially in industrial inspection tasks such as locating the corners and/or edges of parts. As such, it is an important tool for higher-level “intelligent” vision systems. Binary correlation is a form of binary template matching which provides a numerical value corresponding to “degree of fit” rather than an “all or nothing” answer. Commercially-available high-speed image processing systems can readily perform this operation using linear convolvers, but such convolvers are very expensive except for very small kernels. Furthermore, linear convolvers constitute a gross “overkill” for the relatively simple operation of binary correlation. Specialized binary convolvers have been built, but are not part of standard commercial systems.

This paper describes a new pipelined implementation of binary correlation which fits into the standard SKIPSM (Separated-Kernel Image Processing using Finite State Machines) architecture and which can be built using standard ICs costing less than $500 total. The same approach can also be implemented in software, providing an order-of-magnitude increase in speed at no extra cost. Furthermore, this same SKIPSM architecture is highly versatile and programmable, allowing it to be software-reconfigured to perform hundreds of other pipelined image processing operations.

Keywords: image processing, binary correlation, separability, pipelining, real time, implementations, finite-state machines, inspection

1. INTRODUCTION

Binary correlation has long been used for image analysis in general and automated visual inspection and quality control in particular, especially for locating edges or objects or features in images. Earlier papers1, 2, 3, 4, 5, 6, 7 presented a powerful and flexible new high-speed implementation technique, under the general acronym of SKIPSM (Separated-Kernel Image Processing using Finite State Machines), which is capable of performing morphology, template matching, and many other image processing operations. There are two papers in this volume (in addition to this one) which further expand the technique.8, 9 Another related operation, grey-level template matching, can also be done with SKIPSM, and will be described in a later paper. SKIPSM avoids the “brute force” approach used in conventional high-speed hardware — separate standard or custom integrated circuits for each function: multipliers, adders, etc. It can be implemented either in low-cost microchips or in software. In either case, it can provide significant increases in speed and/or neighborhood size.

For the generalized SKIPSM approach, four key ideas are involved:

• the separation of 2-dimensional operations into a row operation followed by a column operation,
• the reformulation of these operations in a recursive (“pipelined”) manner,
• the implementation of these recursive operations as finite state machines, and
• the automated generation of the finite state machine configuration data.

Note that the separation of 2-D operators into two 1-D operators does not involve separability in the usual sense, such as is defined for 2-D linear convolutions. All 2-D operators meeting a simple separability condition1 can be separated using SKIPSM, although the result may be unwieldy in some cases. This condition is met for binary correlation.

Some key features of SKIPSM, specific to binary correlation, are as follows:

• Binary correlation with moderate-sized arbitrary kernels can applied in a single pipelined pass.
• Multiple kernels can be applied simultaneously in a single pass. Example 2 in this paper applies 8 at one time.
• The user specifies the kernel or kernels. All other steps can be automated.

Speed increases and/or neighborhood size increases can be achieved using conventional pipelined hardware in this new way. Alternatively, inexpensive off-the-shelf “chips” can be configured to carry out the same operations as much more expensive conventional image processing hardware. Corresponding “speedups” are achieved in software-based implementations.

2. FSM ESSENTIALS AND SEPARABILITY AS APPLIED TO BINARY CORRELATION

The earlier papers1, 2, 3, 4, 5, 6, 7 include discussions of SKIPSM and finite-state machine (FSM) fundamentals and implementation architectures. This paper builds on the developments given in the binary morphology paper2 and the binary template matching paper,6 the details of which will not be repeated here. The overall conclusions of these papers, as applied to the binary correlation, can be summarized as follows:
The binary correlation operation can be separated into a row operation followed by a column operation. The row operation, called row encoding, can be written in recursive form, so that it can be “pipelined.” This recursive row formulation can be realized as a finite-state machine. Many different row encoders can be combined into a single finite-state machine. Therefore, a single row machine can simultaneously encode all the different row patterns needed by all the kernels being applied. In this paper, a set of “universal” row encoders are used. Each encoder can be used for all kernels of a given width, independent of height. This row machine can be implemented with the SKIPSM architecture (Figure 1) using inexpensive standard microcircuits. The column operation which gives the desired 2-dimensional result can be written in recursive (“pipelined”) form. This recursive column formulation can be realized as a finite-state machine. This column machine can be implemented with the SKIPSM architecture (Figure 1) using inexpensive microcircuits. Many different column machines can be combined into a single finite-state machine. Therefore, a single column machine can simultaneously encode all the 2-dimensional kernels being applied. For given RAM chips, the number of row encoders that can be combined into one row FSM and the number of 2-D kernels that can be combined into one column FSM depend in a complex way on the size, shape, and patterns of the individual kernels. The best way to determine the limits for a given case is to actually compile the data using the SKIPSM software.

### 3. A Family of Template Matching Operations

In general, binary correlation is one of a family of operations in which a “small” reference image, called the kernel or template, is tested and compared for similarity at every possible position within a “large” input image. For pipelined image processing systems, the kernel is moved across the image in raster-scan order. The output pixel at each position is set to a value representing the degree to which matching (according to some criterion) is achieved at that position. These techniques can be used to locate particular objects or features within an image. Two common examples of the use of binary correlation:

- Dimensions or positions of objects can be determined by using kernels to find corners, edges, logos, or other features.
- Particular kinds of defects (e.g., cracks, spots) can be detected and located with appropriately-chosen kernels.

Classification of the operations in this general family can be done according to the kinds of images involved and according to the matching criterion used, as summarized for binary images in Table 1, adapted from the binary template matching paper. Input images and templates can be “binary” or “grey-level” or “color.” This paper is restricted to binary input images. The necessary computational effort increases rapidly as we go from binary to grey-level to color images. Grey-level morphology is covered in an earlier paper. Grey-level template matching will be covered in a paper to be published later.

- Matching can be “exact,” requiring that, in order for the output to be non-zero, every pixel of the image must match exactly with the corresponding pixel of the template at the current position. This gives a binary output: “no match” ⇒ “black”; “exact match” ⇒ “white.” Alternatively, matching can be fuzzy, in that only a specified fraction of the pixels must match exactly in order to give a white. As another alternative, the output can be proportional to the number of matching pixels (whether black or white). This is called binary correlation, and is the subject of this paper.

<table>
<thead>
<tr>
<th>Template/Kernel Type</th>
<th>Name of Resulting Operation (with references to related papers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>all white, or all white plus “don’t care”</td>
<td>binary erosion(^2) and skeletonization(^7,8)</td>
</tr>
<tr>
<td>all black, or all black plus “don’t care”</td>
<td>binary dilation(^2) followed by negation</td>
</tr>
<tr>
<td>black and white, or black and white plus “don’t care”</td>
<td>binary template matching(^6)</td>
</tr>
</tbody>
</table>

Table 1. Classification of types of binary template matching, including binary correlation

### 4. Binary Correlation with No “Don’t Care” Pixels

Binary correlation applies a binary kernel at each pixel in the image and gives as the output at that pixel the number of pixels at which the kernel and the image are identical. Initially it is assumed that there are no “don’t care” pixels in the kernel. This is an important
assumption, because “don’t care” pixels may complicate the problem and lead to additional states in the FSMs. On the other hand, kernels without “don’t care” pixels can be more susceptible to errors due to noise or sub-pixel shifts in image alignment. Therefore, it is important in some cases to be able to handle kernels with “don’t care” pixels.

A series of examples will be used to illustrate these ideas, starting with a very simple 3-pixel by 3-pixel corner detector, as shown in the center of Figure 2. (It is called “SW” because it corresponds to the south-west direction on a standard map.) The left part of the figure shows a section of a binary image to which this kernel is to be applied. On the right the results of applying the kernel at two positions in the image are shown. We will say that the kernel is placed “at \( i,j \)” when the center pixel of the kernel is positioned at row \( i \), column \( j \) of the input image. The kernel is first placed at \( 1,1 \) and the convolution value is computed. The resulting convolution value, 5, is placed at position \( 1,1 \) of the output image. The kernel is then moved to \( 1,2 \) and the resulting convolution value, 9, is placed at position \( 1,2 \) of the output image. Et cetera. Since there are 9 pixels in the kernel, the maximum possible output value for this example is 9. The three places in the image at which perfect correlation occurs are indicated by heavy crosshatched squares.

The actual process of counting the number of pixels which are in agreement can be done in a variety of ways. One simple way of expressing the computation mathematically is as follows: Let black pixels in both kernel and input image be represented by a Boolean TRUE or 1. Then the output value can be written as the sum over all pixels of the kernel of the quantity XNOR(image pixel, corresponding kernel pixel), where the Boolean 0s and 1s resulting from the XNOR (exclusive NOR) operation are converted to numerical 0s and 1s. The actual process of counting the number of pixels which are in agreement can be done in a variety of ways. One simple way of expressing the computation mathematically is as follows: Let black pixels in both kernel and input image be represented by a Boolean TRUE or 1. Then the output value can be written as the sum over all pixels of the kernel of the quantity XNOR(image pixel, corresponding kernel pixel), where the Boolean 0s and 1s resulting from the XNOR (exclusive NOR) operation are converted to numerical 0s and 1s. The actual process of counting the number of pixels which are in agreement can be done in a variety of ways. One simple way of expressing the computation mathematically is as follows: Let black pixels in both kernel and input image be represented by a Boolean TRUE or 1. Then the output value can be written as the sum over all pixels of the kernel of the quantity XNOR(image pixel, corresponding kernel pixel), where the Boolean 0s and 1s resulting from the XNOR (exclusive NOR) operation are converted to numerical 0s and 1s. The actual process of counting the number of pixels which are in agreement can be done in a variety of ways. One simple way of expressing the computation mathematically is as follows: Let black pixels in both kernel and input image be represented by a Boolean TRUE or 1. Then the output value can be written as the sum over all pixels of the kernel of the quantity XNOR(image pixel, corresponding kernel pixel), where the Boolean 0s and 1s resulting from the XNOR (exclusive NOR) operation are converted to numerical 0s and 1s.

Figure 3 shows definitions, the state transition diagram, and a block diagram for a universal row machine that can be used for all 3-column-by-n-row binary correlation kernels, where \( n \) is any value. This row machine has a binary input (1 bit), four internal states (2 bits), and output values 0 to 7 (3 bits). A software implementation requires only 8 bytes (not Kilobytes or Megabytes) of RAM to store two small lookup tables, but carries out the complete row encoding process with only one address computation or concatenation operation and two (RAM or ROM) memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate. (The concatenation of all 3-bit by 5-bit ROM or RAM memory fetches per pixel — one for the next state and one for the output value. The hardware implementation requires only one inexpensive standard chip, and runs in a pipelined mode at the pixel rate.
state and input bits to form the address for the “Next State” and “Output” lookup tables is built into the wiring and requires no additional step.) This machine is so simple that it was created “by inspection” without the need for any automatic FSM generation software. The “Cross-Correlation Table” shown is not part of the row machine. It is built into the column machine, so that this machine “knows” how to interpret the values 0 through 7 received from the row machine. For example, if at a particular moment the column machine is “looking for” Pattern P3 and receives the value 6 from the row machine, it “knows” (because of its built-in knowledge of the cross-correlations) that the correlation between the “desired” pattern P3 and the actual pattern received is 1. It then adds 1 to the partial correlation sum it is forming. Thus, the row machine outputs are actually coded values which convey to the column machine all the information about the relevant input pixel sequence.

Figure 4 shows the state transition table and block diagram for the column machine to implement binary correlation with the “SW” kernel in a pipelined mode. (In this table, “S” stands for “State” and “NS” stands for “Next State.”) Again, the memory or hardware requirements are almost trivial: 128 bytes to store both the “Next State” and “Output” lookup tables.

<table>
<thead>
<tr>
<th>S</th>
<th>In</th>
<th>NS</th>
<th>Out</th>
<th>S</th>
<th>In</th>
<th>NS</th>
<th>Out</th>
<th>S</th>
<th>In</th>
<th>NS</th>
<th>Out</th>
<th>S</th>
<th>In</th>
<th>NS</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>9</td>
<td>0</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td>1</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>9</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>3</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>5</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>7</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>0</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>8</td>
<td>5</td>
<td>10</td>
<td>2</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>12</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>12</td>
<td>4</td>
<td>10</td>
<td>3</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>10</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>11</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>11</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>11</td>
<td>5</td>
<td>11</td>
<td>1</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>11</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>11</td>
<td>5</td>
<td>11</td>
<td>2</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>12</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>14</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>14</td>
<td>4</td>
<td>11</td>
<td>3</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>11</td>
<td>5</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>11</td>
<td>6</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>11</td>
<td>3</td>
<td>11</td>
<td>7</td>
<td>11</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 4. State transition table and block diagram for the 3x3 “SW” column machine.

### 5. Applying Many Binary Kernels Simultaneously

Applying a single 3x3 binary kernel, as in Example 1, is in fact rather trivial. This operation can be done fairly quickly with a variety of hardware components, and has few enough steps to be fairly fast if done conventionally in software. However, in many inspection applications we must apply many kernels. For example, we may wish to find not only “SW” corners but many other types of corners and edges as well. This would require many passes with either conventional hardware or with software, involving either much higher cost or much slower operation.

It is precisely at this point that the SKIPSM approach shows its power. Example 2, shown in Figure 5, applies eight corner detectors simultaneously. The assumption is made for this example that we are interested only in the highest correlation value of the eight being computed. Therefore, the final output consists of two parts: 1) a number from 0 to 7 to indicate the “Winner” — which of the 8 corner detectors had the highest correlation value, and 2) the corresponding correlation value, a number from 0 to 9. Ties are resolved in favor of the kernel with the lowest code number.

Other forms of output are also possible. For example, all 8 correlation values could be provided in parallel. Or the two highest. Or any other specified combination. Since SKIPSM output functions can be specified arbitrarily, any output format could be used. The one chosen for this example is especially good for finding corners, because by the very nature of the problem we are only interested in high correlations, and when the correlation is high for one pattern it is automatically low for the others.
This example also shows another aspect of the SKIPSM paradigm: There is often a large “compression” in the number of states required, and hence in the amount of hardware or software RAM space needed. In this example, six of the detectors have 21 states and two (SW and SE) have 16 states. If these eight column machines ran independently, there could be up to $21 \times 21 \times 21 \times 21 \times 21 \times 21 \times 16 \times 16 = 21,956,126,976$ states. But we have only 73 states for the combined column machine, an amazing reduction of $300,768,862:1$. Furthermore, 73 states is a very small number, as SKIPSM implementations go. (Anything up to 1024 is considered to be normal, and 2048, 4096, 8192, 16384, 32768, or 65536 are possible.) Therefore, we could probably include many other simultaneous kernels — horizontal, vertical, and diagonal edges, for example.

6. The E Kernel, a 5x5 Example

Figure 6 shows a 5-pixel-by-5-pixel binary kernel (which will be referred to as the E kernel), and its application to a simple binary image. This version of the E-kernel also has no “don’t care” pixels. The kernel is first placed at 2,2 and the convolution value is computed. The resulting convolution value, 5, is placed at position 2,2 (row 2, column 2) of the output image. The kernel is then moved to 2,3 (row 2, column 3) and the resulting convolution value, 8, is placed at position 2,3 of the output image. Et cetera. (Computation details for positions 2,2 and 3,2 are shown.) Since there are 25 pixels in the kernel, the maximum possible output value for this example is 25. The Xs in Figure 6 show pixels at which the correlation was 19 or higher (0.76 to 1.00). The left-hand drawing in Figure 7 shows the output image resulting from the application of the E kernel to all of the pixels of the input image, with numerical values representing grey levels. No calculations are shown here for the border pixels (first two and last two rows and columns) of the output image: These pixels can not be “reached” by the kernel because to do so would involve having part of the kernel extend beyond the boundaries of the input image. These border pixels can be handled in various ways. The details are omitted here.

Output pixels containing local maxima are indicated by heavier outlines in Figure 7. Three of these (with values 25, 23, and 22, corresponding to correlations of 1.00, 0.92, and 0.88, respectively) are truly significant, represent perfect or good fits. The other local maxima have much lower peak values (i.e., poor fit). Furthermore, local maxima at the edge of the processing region are usually not true maxima, and are often discarded.
The next step in the use of correlation techniques is usually to examine only the points of high correlation using appropriate logical rules. These tests typically cannot be done in a pipelined system, and therefore a CPU or DSP chip with random access to the image buffer is required (implication: “slow”). But in virtually all applications, only points of high correlation (typically above about 0.9) are of interest. Points of low correlation should be eliminated from consideration, if possible, to save CPU time. The right-hand diagram of Figure 3 shows the result of subtracting a constant (in this case, 18, corresponding to a correlation of 0.72) from all the “raw” correlation values, and setting negative values to zero. The values and x-y locations of the non-zero pixels in this zero-suppressed image can easily be written to a data buffer in real time by a pipelined feature extraction module, thus greatly reducing the searching task of the CPU and saving a great deal of time.

This is something that should be done in almost all (if not all) correlation computations. But it also provides a significant benefit to the FSM implementation: The most important limiting factor in most SKIPSM applications is the number of states. A secondary factor is the number of output levels. In this simple example, suppressing low (and therefore uninteresting) correlation values can
reduce both of these numbers. When multiple FSMSs are being combined into a larger overall FSM (as in Example 2 above), this kind of reduction may be important. Finally, the nonlinear function involved in suppressing the low values is easily implemented in a FSM with no increase in complexity. (FSM outputs are always specified arbitrarily anyway, so this function is easy to implement as any other.)

Figure 8 shows two row machines that could be used for the E kernel: a “universal” row machine for kernels five pixels wide, and a row machine “stripped down” to the three different kinds of rows appearing in this kernel. The “stripped” row machine will usually have fewer output values and therefore may require fewer bits in hardware implementations. In this example, both row machines have 16 internal states, the universal machine has 32 output levels, and the stripped machine has 18 output levels. In either case, 5 output bits are required and there appears to be no advantage in going to the stripped machine. But the resulting column machine, because it has a smaller number of inputs, can be packed into a smaller space. Otherwise, the functioning is identical.

The lookup tables in Figure 8 use the “deficit mode.” In this method of calculation, the correlation deficits — the amounts by which the correlation falls short of perfect correlation at each stage — are accumulated by the column machine, rather than the correlation values themselves. At the end of the process, these are converted back to ordinary correlation values simply by subtracting the deficit from the value for perfect correlation. This method is used because it makes it very easy to limit the number of output levels in recursive calculations: When the accumulated deficit equals or exceeds the number of levels specified, the deficit is simply “clamped” at the maximum value, and no further accumulation takes place.

Figure 9 shows the overall SKIPSMS implementation for the E kernel, with the number of output levels set at six (correlations ≥ 0.80). This fits easily within the “standard” 1024-state SKIPSMS architecture, as defined in earlier papers.1, 2, 3, 4, 5, 6 The critical limiting factor here is the number of column-machine states. This number varies with the number of output levels as follows (and as shown in Figure 10):

<table>
<thead>
<tr>
<th>Number of Levels</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of States</td>
<td>5</td>
<td>11</td>
<td>25</td>
<td>58</td>
<td>135</td>
<td>271</td>
<td>515</td>
<td>884</td>
<td>1368</td>
<td>1924</td>
</tr>
</tbody>
</table>

Any one of these can be implemented using SKIPSMS techniques. However, the “standard” SKIPSMS architecture can implement only up to 9 output levels in this case because it is designed for up to 1024 states (10 bit feedback path). This 10-bit path was chosen so that a certain standard 10-bit programmable delay line could be used. Simply using other chips, or two of these chips, would permit many more states. Therefore, this is not a fundamental limitation of SKIPSMS. For software implementations, this is not a problem.

The graph on the left in Figure 10, which applies only to the E kernel example, shows the effect (lower curve) of suppressing low correlation values as a function of the number of output levels retained. The two-level case corresponds to exact binary template matching. Four, five, or six levels (i.e., retaining correlations ≥ 0.88, 0.84, or 0.80) would be appropriate for typical 5x5 correlation
5. Binary Correlation with “Don’t Care” Pixels

In some cases we may wish to use a binary kernel with some “don’t care” pixels. This allows for a certain amount of “raggedness” (due to thresholding, noise, etc.) around the edges of an imaged object. “Don’t care” pixels add some complexity to the problem, but the same overall approach can still be applied. Figure 11 shows the $E$ kernel modified to include five “don’t care” pixels (crosshatched). In either deficit-mode or normal-mode calculations, these pixels are ignored and therefore never add to the deficit, and the “target” for perfect correlation is lowered accordingly (from 25 to 20 in this example). Other than this, the procedure is essentially the same. Row machines for this kernel are documented in Figure 13.

6. Procedure for Automatic Generation of State Transition Tables

A routine procedure will now be given for designing FSMs for specified kernels. Please refer to earlier papers1, 2, 3, 4, 5, 6 for more details. First, a fundamental concept underlying all SKIPSM applications will be presented — the fact that the program for generating the lookup table for an on-line finite state machine is itself the desired finite state machine. (See Figure 12.) In fact, if this FSM and

---

**Figure 10.** The effect of suppressing low correlation values for the $E$ kernel.

Left: maximum possible states (upper curve) and actual states (lower curve). Right: compression ratio (maximum/actual) problems. The results for adding additional levels are shown to illustrate the powerful effect this parameter has on the number of states, thereby motivating the suppression of unnecessary output levels. The figure also shows the number of states that would be required if there were no “state compression” taking place (upper curve). The graph on the right shows the “compression ratio” — the ratio of the number of actual states to the maximum number of states possible in a combinatorial sense. This sort of compression is highly satisfactory from an implementation standpoint, and is typical of SKIPSM implementations.

---

**Figure 11.** The $E$ kernel with five “don’t care” pixels.

---

**Figure 12.** An iterative procedure for generating SKIPSM lookup tables.
the resulting SKIPSM implementation were placed in separate black boxes, there is no logical or mathematical test that an outside observer could use to distinguish them, except perhaps for their speed of operation and power consumption. All the (by now) thousands of applications and examples of SKIPSM implementations have used the identical iterative lookup-table generation procedure — i.e., the same spreadsheet macros or the same C programs. These examples differ only in the computational or logical rules being applied by the two software-based FSMs at the heart of the procedure — one for the row machine and one for the column machine. For example, binary morphology uses logical ANDs and ORs. Column or row integration uses arithmetic summation. Blob filling uses a logical toggle. And the application covered in this paper, binary correlation, uses summation of pixel-by-pixel correlation values, with or without limiting (zero suppression).

To give more insight into this process, the row machine for the $E$ kernel with five “don’t care” pixels (as in Figure 11) will be presented in some detail. Figure 13 shows the 32 possible 1x5 input sequences, numbered in binary order ($black = 0$, $white = 1$). The State for this machine is defined as the binary value of the four most recent pixels, excluding the current input. The Next State drops the oldest pixel and adds the current input pixel to the set. Two forms of Output are shown: For the “universal” row machine, the pattern number is used as the output, giving 32 output levels. Or, for the “stripped” machine, the indicated coded value is used, giving a reduction to 24 output levels. This reduction is possible because there are only 24 unique combinations of the three correlation deficit values for $P0$, $P1$, and $P2$. (For example, note that patterns 6 and 23, while obviously different, give the same three values of correlation deficit.) These combinations are assigned numerical values from 0 to 24. Any assignment scheme can be used. The resulting values, and the corresponding deficit values, in the form of an input lookup table, are used in the generation of the corresponding column machine, so that the generating program “knows” exactly what correlation deficits are associated with each input value (input “code”) it receives and can construct the column machine accordingly.

The diagrams at the right of Figure 13 show two examples of how the Next State and the Output are computed from the State and the Input at each moment for the row machine. A similar iterative approach is used for the column machine, with the correlation deficits provided by the row machine being accumulated (and perhaps limited to the specified number of levels) as the column machine “marches” down the columns. For further details, please see the earlier papers.1, 2, 3, 4, 5, 6

---

Figure 13. Row encoder details for the $E$ kernel with five “don’t care” pixels.
9. Conclusions

For many important image processing and image analysis problems, appropriately-designed finite state machines can provide very efficient and inexpensive implementations capable of operating at video rates. This paper has described the application of SKIPSM to binary correlation. Arbitrary kernels or many simultaneous kernels can be applied using readily available microchips.

As mentioned here and shown in detail in the companion papers, the difficulties involved in the design of complex FSMs have been overcome, at least in the case of binary morphology and binary correlation, by the development of automated lookup table generation algorithms.

Finally, simple custom or semi-custom hardware implementations designed for these functions can provide binary correlation operations at a fraction of the cost for the corresponding conventional hardware.9

10. References