SKIPSMS:
SEPareted-Kernel IMAGE PROCESSING USING finite-STATE Machines
Frederick M. Waltz, 2095 Delaware Avenue, Mendota Heights, MN 55118-4801 USA

ABSTRACT
This paper presents a fundamentally new way to carry out many standard image processing operations. In comparison with conventional hardware-based and software-based approaches, SKIPSMS (Separated-Kernel Image Processing using Finite State Machines) allows implementation at higher speeds and/or lower hardware cost. The key features of SKIPSMS are:

- the separation of a large class of neighborhood image processing operations (generally considered not to be separable) into a row operation followed by a column operation,
- the formulation of these row and column operations in a form compatible with pipelined operation,
- the implementation of the resulting operations as simple finite-state machines, and
- the automated generation of the finite-state machine configuration data.

Speed increases and/or neighborhood size increases by factors of 100 or more are 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 conventional hardware. Corresponding “speedups” are achieved in software-based implementations. Furthermore, it is often possible to use SKIPSMS to carry out 10 or more different image processing operations simultaneously, with no additional processing steps or hardware.

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

1. INTRODUCTION

Certain image processing operations have become “standard” in the field: linear convolutions, minimum and maximum, morphology, and so on. The techniques for implementing these operations have largely become standardized also. Where the speeds available from affordable software-based systems are insufficient, a number (anywhere from one to dozens) of hardware boards are connected together in various ways to perform desired combinations of operations on the input images. The conventional approach to carrying out image processing operations in fast hardware is essentially a “brute force” approach, using separate standard or custom integrated circuits for each function: multipliers, adders, random-access memories, and so on. Improved performance (e.g., larger neighborhoods, faster speeds) is achieved by using more and/or faster functional blocks. The observed trend of ever-increasing performance for such boards is due almost exclusively to the continuing miniaturization of the integrated circuits used, and not to any new algorithms or ways of implementing the desired operations.

In contrast, this paper and the companion papers1, 2, 3, 4, 5 present what is believed to be a completely new way to implement a variety of important image processing operations. The acronym SKIPSMS (Separated-Kernel Image Processing using Finite State Machines) has been given to this family of techniques. This approach can use existing hardware, or can be implemented in custom hardware, or can be implemented in software. Whatever the implementation, greatly improved performance is obtained, which may take the form of increased speed or larger neighborhoods or both. As new boards and components with enhanced performance become available, these new techniques will provide comparable additional performance improvements. Stated another way: Whatever components are needed to achieve a specified result using conventional implementations, the use of these new techniques (where they apply) will allow either increased performance with the same components, or the same performance with fewer components, or some combination of these two effects.

The image processing categories to which SKIPSMS can be applied include but are not limited to the following:

- Binary morphology of all types with large, arbitrary SEs (structuring elements). SEs up to 25x25 and larger and with “holes” and other non-convex shapes can be applied in a single pipelined pass.
- Multiple SEs can be applied simultaneously in a single pipelined pass. For example, 6 or more stages of the “Grassfire Transform” have been carried out in one pass, and even larger numbers of stages are possible.
- Grey-level morphology, if the number of grey levels is not too large.
- Binary template matching with large, arbitrary templates.
- “Fuzzy” binary template matching and binary correlation.
- Grey-level template matching, if the number of grey levels is not too large.
- Various “smearing” operations, including row, column, and diagonal summations and Hough transforms.
- Certain operations previously thought to be impossible in a pipelined system, such as “blob fill” and “patterned blob fill.”
- Pipelined generation of certain standard images (e.g., grey-level wedges) and arbitrary image patterns (textures).
This paper presents the fundamental ideas of SKIPS. Details of many of the above-noted applications are given in the companion papers.\textsuperscript{1, 2, 3, 4, 5}

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 condition (defined below) can be separated using SKIPS, although the result may be unwieldy in some cases.

Finite-state machine theory is a branch of automata theory. Finite-state machines (henceforth referred to as FSMs) are not machines, in the usual sense, but autonomous input-driven sequential mathematical/logical constructs having important analytical and computational properties and a very well-developed body of theory. Hardware implementations of FSMs using flip-flops, multiplexing switches, or ASICs are widely used for such things as sequencers, timers, and computer disk I/O drivers.

The use of FSMs to “trap” certain patterns of peaks and valleys in 1-dimensional data arrays derived from 1-D DOLP (Difference-of-Low-Pass) transforms of signals from line-scan cameras was described in an unpublished technical report.\textsuperscript{6} This did not involve image processing, but a feature extraction operation one step removed from image processing. As for other applications to image processing, my examination of thousands of books and articles on image processing has revealed no direct references to FSMs. Furthermore, judging by the available products, high-speed hardware vendors make no use of FSMs for image processing. Finally, FSMs are not part of the usual training given to practitioners in the image processing field.

2. The Separation of “Non-Separable” Two-Dimensional Operators

Many neighborhood (i.e., 2-dimensional) operators, such as a binary morphology, binary template matching, grey-level morphology, area summation, neighborhood maximum and minimum, etc., can be separated into 1-D row and column operations. Note first that the overall 2-dimensional result involves some kind of test made on each pixel, as well as functions for combining the individual test results to determine the overall output value. Definitions:

- Let K and L denote the numbers or rows and columns, respectively, in the neighborhood, kernel, structuring element, etc.
- Individual pixel function $q_{ij}, i = 1, 2, \ldots, K; j = 1, 2, \ldots, L$: This function has no direct bearing on the question of separability. For some operations, such as neighborhood maximum and neighborhood minimum, $q_{ij}$ is just the pixel value itself. For others, it is some logical or arithmetic function of the pixel value and the corresponding structuring element or kernel value. Examples: For binary erosion, let the pixels which are part of the structuring element (SE) be assigned a Boolean value of 1, and the “don’t care” pixels a Boolean value of 0. Similarly, let the image pixels be assigned values of Boolean 0 for “black” and Boolean 1 for “white.” Then, for a given placement of the SE on the image, the (Boolean) result of the pixel test can be stated as $q_{ij} = (\text{image pixel value}) \lor \neg (\text{SE pixel value})$. For grey-level erosion, $q_{ij} = (\text{image pixel value}) - (\text{SE pixel value})$. For logical or arithmetic functions, $q_{ij}$ is just the pixel value itself. For others, it is some logical or arithmetic function of the pixel value and the corresponding structuring element or kernel value.
- Overall function $F$: For each position of the neighborhood on the image, this uniquely calculates the overall output directly as a function of the pixel results. Thus, overall output $= F(q_{11}, \ldots, q_{1L}, q_{21}, \ldots, q_{2L}, \ldots, q_{K1}, \ldots, q_{KL})$.
- Row combining function $R$: For each row, this calculates a row result as a function of the pixel results for that row.
- Column combining function $C$: This calculates an overall output result $C$ from the set of row results $R$.

Separability condition: Separability requires that the overall result be computable from the results for the individual rows. Thus, there must exist row and column functions $R$ and $C$ such that

\[
F(q_{11}, \ldots, q_{1L}, q_{21}, \ldots, q_{2L}, \ldots, q_{K1}, \ldots, q_{KL}) = C[R(q_{11}, \ldots, q_{1L}), R(q_{21}, \ldots, q_{2L}), \ldots, R(q_{K1}, \ldots, q_{KL})]
\]

Revision note – In later papers in this series, this is called the compressibility condition. This change was made for two reasons:

1. All two-dimensional operators are separable, in the sense that row machine in the form of a serpentine memory can be used to provide, at each moment, all the pixel data needed by the column machine. Furthermore, this serpentine memory can easily be realized as a finite-state machine. (Proof omitted.)

2. When a serpentine memory is used, there is no data compression and many of the advantages of SKIPS are lost. However, when the above condition is met, some data compression, often a large degree of data compression, takes place. Therefore, the new name is more appropriate and more informative.

A sufficient condition for separability is that the set of values produced by the pixel functions $q_{ij}$ and the overall combining function implied by $F$ form a group. (Actually, only the closure and associativity properties of a group are required.) Most of the operators used in image processing meet this condition: binary erosion and binary template matching (function = Boolean AND), image

SKIPSM: Separated-Kernel Image Processing using finite-State Machines

SPIE Paper # 2347-36

F M Waltz

SK1-2
addition and convolution (function = add), neighborhood maximum or minimum (function = maximum or minimum), and grey-scale erosion (function = minimum of differences). Note that the row and column combining functions R and C need not be the same. Thus, operations such as the minimum of the row maximums can be separated.

One operation that does not meet the requirement is the ranked filter operation, for ranks other than maximum or minimum. Thus, the median of the row medians is, in general, not the median over the whole neighborhood. Therefore, the ranked filter operation cannot be separated in the way described here. (But see 7 for a partial solution to simplified implementations of ranked filters.)

The separation process can best be understood by considering it in a particular application. Binary erosion has been chosen for illustrative purposes, but a similar approach applies to other operations. Consider the binary structuring element (SE) of Figure 1 in the position shown on the image. The requirement for not eroding the center pixel of the output image is that, for a given placement of the SE, the image must have a “white pixel” everywhere that the SE has a “white” pixel. The image may be either “white” or “black” at the other “don’t care” SE pixel positions. For the first row of the SE, the results are \([1, 1, 1, 1, 1]\), where 1 implies a Boolean 1 for the “don’t care” case (at which the condition is always satisfied), 1 is a Boolean value implying that the test is satisfied there, and 0 is a Boolean value implying that the test is not satisfied.

The row result for the first row of the SE is then the Boolean value \(1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 1\), where \(\cdot\) represents the Boolean AND operator, which is of course associative. For all five rows of the SE, the results can be written thus:

<table>
<thead>
<tr>
<th>SE Row</th>
<th>Individual Pixel Results</th>
<th>Row Output Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 1 1 1 1</td>
<td>(1 \cdot \cdot \cdot \cdot = 1)</td>
</tr>
<tr>
<td>2</td>
<td>1 0 1 1 1 1</td>
<td>(1 \cdot 0 \cdot 1 \cdot 1 \cdot 1 = 0)</td>
</tr>
<tr>
<td>3</td>
<td>0 1 1 1 1 1</td>
<td>(0 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 0)</td>
</tr>
<tr>
<td>4</td>
<td>1 1 1 1 0 1</td>
<td>(1 \cdot 1 \cdot 0 \cdot 1 \cdot 1 = 0)</td>
</tr>
<tr>
<td>5</td>
<td>1 1 1 1 1 1</td>
<td>(1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 1)</td>
</tr>
</tbody>
</table>

The final 2-D output value is then the five row values ANDed together, thus: \(1 \cdot 0 \cdot 0 \cdot 1 \cdot 1 = 0\). Therefore, the SE does not “fit” onto the image (i.e., meet the conditions) at this point, and the overall output is 0 (“black”). This result can obviously be obtained by ANDing together all 25 pixel values (or all 18, if the “don’t cares” are omitted). But it can also be obtained by first obtaining the five row values and then ANDing them together. This seems trivially obvious, but many important results follow from this simple fact, as will be seen below and in the companion papers. Figure 2 shows two versions of the block diagram for this kind of separated 2-D operation.

For the remainder of this paper, discussion will be limited to the first configuration shown in Figure 2. The other configuration can be handled in an analogous manner.

3. Recursive Formulation of the Separated Operators

There is much to be gained if the row and column functions R and C can be “pipelined” (i.e., calculated recursively). Most functions that can be separated can also be calculated recursively, including all those claimed to be separable in

Figure 1. An example of the separation of a 2-D operation.

Figure 2. Two versions of the basic SKIPSM architecture for a row-by-row raster scan pattern.
Section 2 above. The conversion of a static or conventionally-pipelined formulation into a recursive formulation will be illustrated for a column operator. The development for row operators is similar and will not be presented here.

| Recursion conditions on row and column combining functions R and C: |
|:------------------------|------------------------|
| The row function R must be expressible in terms of a dyadic function \( r(a,b) \) which is associative. That is |
| \( R(q_{i1}, q_{i2}, q_{i3}, q_{i4}, \ldots, q_{iL}) = r(\ldots r(r(q_{i1}, q_{i2}), q_{i3}), q_{i4}), \ldots, q_{iL}) \) |
| The column function C must be expressible in terms of a dyadic function \( c(a,b) \) which is associative. That is |
| \( C[R_1, R_2, R_3, R_4, \ldots, R_K] = c(\ldots (c(c(R_1, R_2), R_3), R_4), \ldots, R_L) \) |
| where \( R_1 = R(q_{11}, \ldots, q_{1L}), R_2 = R(q_{21}, \ldots, q_{2L}), \ldots, R_L = R(q_{K1}, \ldots, q_{KL}) \) |

Consider Figure 3, in which a neighborhood operator is placed on a section of the input image in five positions. Below each diagram are the five conditions \( R_i(J) \) that determine the overall output at that time by means of the column combining function C. Here \( R_i(J) \) is the row result obtained by applying the pixel tests required by row \( i \) of the SE to the corresponding pixels of row \( J \) of the image. The row result \( R_i \) and the overall output \( C \) may be Boolean or numeric, depending on the operations involved.

Now assume that we are receiving the input image in “pipelined” raster-scanned order and that we have just received the pixel from line \( U \) corresponding to the lower right corner of the neighborhood. We must now calculate the output value \( F (= C) \). If we were operating using a conventional “brute force” pipelined architecture, we would receive 25 pixel values previously placed in a serpentine memory buffer, make 25 pixel tests (or 18 tests, if “don’t care” tests are omitted), and combine these 25 tests according to rule \( F \). At each successive pixel time we would do all these steps again. There is no way to eliminate the need to make all 25 tests at each position. But the recursive formulation eliminates the serpentine memory and retains the results of past calculations rather than the raw image pixel values. Depending on the operation, this may require considerably less hardware than the conventional approach, which becomes highly impractical for neighborhoods with many rows. (SKIPSIM, using only an inexpensive 10-bit delay line and a few RAM chips, can easily handle binary morphology with 25x25 structuring elements or larger. I know of no “brute force” implementation, regardless of cost, that comes even close to this.) For some other operations, such as linear convolution, the recursive formulation requires the storage of large amounts of information and therefore should not be used.

In Figure 4, the same five sets of results shown in Figure 3, along with additional results derived from the same image for additional downward shifts of the test neighborhood, are listed in columns. The information needed to make the output calculation at time \( U \) is...
shown in the first column, enclosed in a dotted box. We note that the output involves the test \( R_5(U) \) made right now on the current row of the image and four previous row results. Thus, it is clear that a recursive formulation must somehow retain these previous four results, since it no longer has access to the earlier raw pixel values.

Figure 4. Information needed for recursive algorithm.

What else must be retained? If we extend this reasoning to the next time and all succeeding times, we see that at time \( T \) we must also have retained the results in columns 2, 3, and 4, as enclosed by the solid-line box. We will call this information the state at time \( T \). At time \( U \), we must retain the results enclosed in the dotted box. And so on.

We can now state the general procedure for the recursive 5-row column operator. For clarity, it will be stated in terms of the values at time \( U \), with the understanding that analogous steps are carried out at each succeeding time:

1. Calculate \( R_5(U) \). Compute the output \( C \) from \( R_5(U) \) and the history \( [R_1(Q), R_2(R), R_3(S), R_4(T)] \).
2. Update the three-stage history \( [R_1(R), R_2(S), R_3(T)] \) to a four-stage history by appending \( R_4(U) \).
3. Update the two-stage history \( [R_1(S), R_2(T)] \) to a three-stage history by appending \( R_3(U) \).
4. Update the one-stage history \( R_1(T) \) to a two-stage history by appending \( R_2(U) \).
5. Create a new one-stage history \( R_1(U) \).

Note that all of these steps depend only on the history (the state) and on results computed from image row \( U \). No previous pixel values are needed at this time.

It would appear that there has been no real gain, because of all the past results that must be remembered. It is at this point that the recursion conditions become important: Under these conditions, a single number can be used to represent the four-stage history, another the three-stage history, etc. If we let \( K \) denote the number of rows in the neighborhood, then the \( K(K-1)/2 \) pieces of information in the state are reduced to \( (K-1) \) pieces of information — a reduction from \( O(K^2) \) to \( O(K) \). If, for example, there are 25 rows \( (K = 25) \), then the 300 pieces of required information are reduced to only 24.

How many bits does it take to represent each of these pieces of information in the history? This depends on the operation. For binary template matching and binary morphology, only 1 bit is required. Figure 5 illustrates this point, as applied to the same example used in Figures 3 and 4.

Figure 5. Reduced state representations for a binary erosion operator.
As will be explained below and in a companion paper,\(^1\) even further reductions in the number of bits required to represent the state are almost always possible. The actual number may be much smaller, depending on the particular problem, and may be as small as \(\log_2(K)\). Thus, the word length to represent the state for a 25-row binary erosion problem may sometimes be as small as \(\log_2(25) = 4.644\), implying a 5-bit state variable. This massive compression (from 300 bits for the “brute force” approach to 5 bits for this example) is one of the primary advantages of SKIPSM.

4. ESSENTIAL ELEMENTS OF A FINITE-STATE MACHINE

Finite-state machines can be implemented in a wide variety of ways. All implementations share these features:

- A variable called the state of the system. It may be a scalar or a vector variable. To qualify as a finite-state machine, the system must have a finite number of distinct states. That is, the state variable can take on only a finite number of distinct values. The definition of the state variable is usually the most important part of the design of a FSM.
- One or more input variables, which cause the system to move from one state to another.
- One or more output variables, which depend only on the current state and the current input.
- Some kind of memory, to allow the system to remember its previous state.
- A full definition of the state transitions, giving the next state and the output in terms of the current state and the current input. This can be in the form of a table, or a set of software operations, or a hardware circuit (gates, adders, etc.).

According to this generalized definition, most if not all pipelined hardware boards are FSMs. For example, a linear convolver with a serpentine memory is a finite state machine. The history (i.e., state) is kept in the serpentine memory. (For 8-bit data, a big serpentine memory involves an extremely large number of states, and should not be thought of as a possible application for SKIPSM.) The multipliers compute the results for individual pixels and the adder computes the overall output. But convolvers are not FSMs in the spirit of SKIPSM, because they use specialized hardware to carry out their functions, and they are reprogrammable only in very limited and predefined ways. In contrast, the SKIPSM architecture is the same for a wide variety of operations, and is highly reprogrammable. A SKIPSM configuration can even be programmed to perform, in parallel, two (or 10) totally unrelated operations on the same image.

Figure 6 shows the implementation of binary erosion with an 11-by-13 rectangular SE using two pipelined (i.e., recursive) finite state machines in sequence. This very simple example — almost trivial for the SKIPSM approach — has been chosen deliberately to simplify the explanation of the various parts of a finite-state machine. For each of the two machines (row and column), the results are given in two forms: a state transition table, and a state transition diagram. These contain exactly the same information, and are completely equivalent. The table is the usual working tool, but the diagram can sometimes help the user understand FSM behavior and design limitations.

The row machine will be discussed here in terms of its state transition diagram. This machine has eleven states, numbered 0 through 10. The zero state, indicated with a heavier border, is called the null state and is usually assumed to be the initial state at the start of each row (for the row machine) or each pipeline pass (for the column machine). Because this is a binary erosion problem, there are two possible input values, 0 (“black”) and 1 (“white”). As a succession of inputs arrive, the state of the machine changes in accordance with the diagram. Arrows between states are labeled with the corresponding input value. To simplify the diagram, transitions back to the null state are omitted here. In this example, there is only one situation that produces a non-zero output: state = 10, input = 1, in which case the output is 1. (Non-zero outputs are shown by a “slash” and then the output value. Zero outputs are omitted here to simplify the diagram.) When the output value is 1, it implies that the most-recent 11 image pixels are all “white” and thus that the row condition has been satisfied. This information is then passed to the column machine, which “counts rows” to determine when the whole structuring element “fits” into the image.

As an exercise, let us consider the behavior of the row machine in response to an input sequence. Shown below are the assumed input sequence and the resulting state sequence and output sequence. The initial state (boldface) is assumed to be zero.

| Input:   | 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 |
| Resulting State: | 0 1 2 0 1 2 3 4 5 6 7 8 9 1 0 10 10 10 0 1 0 0 |
| Resulting Output: | 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 |

The crucial point: Because this is a recursive implementation, the machine can not simply start over counting up from zero again, once it has found the required 11-in-a-row situation. At the very next pixel time, it must produce an output telling whether the new set of 11 most-recent pixels is “all white” (1) or “not all white” (0). All of this “bookkeeping” is taken care of automatically by the FSM. More complex examples in the companion papers show less trivial behavior.

Additional comments: This example is so simple that, with a little experience, one can write down the answer by inspection. More difficult problems give much more complex diagrams, which can not be done without appropriate FSM software. The inexpensive
A very simple binary erosion operation implemented as two finite state machines.

RAM implementation shown in Figure 6 produces the desired result in one pass. This operation would require six passes if conventional (and much more expensive) 3-by-3 pipelined morphology hardware were used. The RAM-based implementation shown at the bottom of Figure 6 are discussed in the next section, along with other implementations.

5. ARCHITECTURES FOR IMPLEMENTING SEPARATED FINITE-STATE MACHINES

FSMs can be implemented in many ways. (See any standard logic design textbook.) The emphasis here will be on RAM- or ROM-based implementations. Figure 7 shows some possibilities.

The distinction between rule-based and RAM-based implementations has many implications, which will be discussed in more detail below (Section 6) and in the companion papers. The RAM chips can be replaced by ROM chips for non-programmable applications.
Flash RAM may be an even better choice for some systems. If the RAM chips do not include clocked latches, then latch chips must be added on the RAM output lines of the FSMs. The additional RAM chips on input and output ports (last two diagrams) allow for input and output lookup tables for thresholding, input grey-level limiting, mapping of simultaneous SKIPSM outputs into appropriate bit planes, etc.

Another alternative is to use commercially-available pipelined image processing boards. Figure 8 shows a system called DART which was designed with such implementations in mind. It is versatile and very flexible, and includes hundreds of image processing operations of all types. Included among these are a wide variety of FSM-based functions.

Unfortunately, this system and all other commercially-available board-level systems I have investigated have one FATAL flaw with respect to FSM implementations: They have latencies of two or more pixel times on all paths between modules. The newest systems (MaxVideo 200 or equivalent) have similar latencies on the internal paths. They can implement column FSMs very well, BUT THEY CAN NEVER IMPLEMENT ROW FSMs, because row FSMs require that the previous state be returned to the LUT input AT THE NEXT PIXEL TIME. Therefore, the row function must be done in some other way or by one of the other boards. The SNAP board (nonlinear 3x3 or 1x9 neighborhood board) can be used for this in many cases, but this limits the neighborhood width to 9 pixels. In combination with the convolver (VFIR) used as a “bit stacker” and the MaxSP LUT as a combiner, widths up to 27 pixels have been demonstrated.

It should be relatively easy for the manufacturers to provide a fast feedback path between the output and the input of a big lookup table, so that both row and column FSMs could be implemented. So far they have not seen fit to do so. Until they do, the only alternatives are to develop custom hardware with a fast feedback path, or to continue to “trick” the existing hardware into doing things that are thought to be “impossible” for it. Ideally, someone will develop a SKIPSM daughter board for the Datacube MaxVideo 200 system, thus freeing that otherwise-excellent system from its design limitations and at last allowing truly powerful and versatile use of important but generally unavailable operations, especially binary morphology.

One other point: Note that software implementations of SKIPSM provide the same kinds of performance improvements for software-based systems as they do for the hardware-based systems that are the primary focus of this paper. In a single memory access, the CPU can obtain the result that might take hundreds of CPU operations if done conventionally. In a very rough way, this is comparable to the difference between interpretive computer languages and compiled languages. One can think of SKIPSM as a method for compiling many CPU operations into one memory-fetch step. The lookup-table implementation requires more memory than the CPU implementation, but since RAM chips are the cheapest and most highly optimized electronic parts, this much higher speed can be obtained at low cost.

6. Creating Lookup Tables for RAM-Based FSMs

For RAM- or ROM-based FSMs, an essential step is the generation (off line) of the lookup tables (LUTs) to be loaded into the RAM chips. The last two columns in the tables for the row machine and the column machine of Figure 6 are precisely the values to be loaded into the “Next State” and “Output” lookup tables for such implementations.
A Procedure for Automatic FSM Lookup Table Synthesis: A lookup-table version of the FSM can be created by

- setting up a rule-based version of the FSM (either in hardware or in software)*
- applying all possible input values to this FSM, initialized to the null state
- applying all possible input values to each new state created as a result of this step
- continuing in this way until no new states are created
- tabulating the resulting states and outputs and sorting them into the desired order
- renumbering the states in consecutive order

Automated table generation procedures based on this procedure have been developed for binary morphology, binary template matching, run-length encoding, blob filling, and certain grey-level morphology problems. The details of automated table generation are left for the companion papers1, 2, 3, 4, 5.

7. Initializing Finite-State Machines

Pipelined boards and custom pipelined hardware typically process everything that comes to them, without “knowing” or “caring” whether the information comes from the active video region, the horizontal retrace interval, or the vertical blanking interval. Thus, unless something is done to prevent it, some FSMs may not start out at the zero state and will “wrap around” from one row to the next, resulting in totally meaningless output. In general, this problem can be solved by using pixel switching (“MOSCs”) in boards such as the Datacube MaxMux, MaxSP or MaxVideo 200, with a mask of 1s where the FSMs should operate and 0s elsewhere. The pixel switching function can be used to force the FSM state and FSM output to zero whenever the mask is zero. Such masks can be provided by the Datacube ROI-Store modules. On the other hand, many FSMs reset themselves after receiving a sufficient number of zeros (or other specified constant). This is particularly true of neighborhood operations (template matching, morphology) and also for those run length encoding problems in which “black” runs are ignored. In these cases, the black area surrounding the image takes care of the resetting, unless the neighborhood is so large as to span the blanking or retrace intervals. If custom hardware is used, care must be taken to see that the FSMs are reset at the appropriate times. Ideally, a global hardware reset or reset sequence for all state registers would be included in the hardware design, as well as individual reset lines for portions of the system.

8. 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 report has shown FSMs being implemented in a variety of ways. In particular, the DART system of pipelined video-rate image processing boards has been designed to implement both very large FSMs and FSMs of small or moderate size. In some cases, DART or other simple board configurations can execute these operations faster by one or more orders of magnitude than the same boards could if used conventionally (i.e., in the ways anticipated and planned for by their designers).

As mentioned here and shown in detail in the companion papers, the extreme difficulties involved in the design of complex FSMs have been overcome, at least in the case of binary template matching and binary morphology, by the development of automated lookup table generation algorithms. Simple custom or semi-custom hardware implementations designed for these functions could provide important image processing and image analysis functions at a small fraction of the cost for the corresponding standard hardware.

9. References


SKIPS: Separated-Kernel Image Processing using finite-State Machines

SPIE Paper # 2347-36

F M Waltz

SK1-9