single match takes, say, one millisecond of CPU time, matching against a database of one million fingerprints would require a total of 10^3 seconds of CPU time. If we have to process 100 queries per day, we would need 10^5 seconds or 27.78 hours of CPU time alone, not including the I/O time in reading the fingerprints from the database.

In order to provide a reasonable response time for each query, commercial systems use dedicated hardware accelerators or application-specific integrated circuits (ASICs). While application-specific architectures and ASICs have been designed to meet the computing requirements of complex image processing tasks, such designs have the following two major limitations: (i) once fabricated, they are difficult to modify; and (ii) the cost of building special-purpose application accelerators is very expensive for low-volume applications. Both of these limitations have been the driving force behind the design of custom computing machines (CCMs) using reconfigurable logic arrays known as field-programmable gate arrays (FPGAs). An attached processor built with FPGAs can overcome the two limitations noted above. High performance is achieved with FPGAs by exploiting an important principle: most of the processing time of a compute-intensive job is spent within a small portion of its execution code [3], and if an architecture can provide efficient execution support for the frequently executed code, then the overall performance can be improved substantially. Portions of the matching algorithm have been identified for implementation on Splash 2, leaving the remainder to be implemented using software on the host.

The goal of this chapter is threefold. First, it describes a successful application using Splash 2. Second, we demonstrate that a suitable mapping of an algorithm to a given architecture results in excellent performance. Third, we illustrate how FPGAs can facilitate this mapping process without sacrificing speed and flexibility. In fact, FPGAs offer greater flexibilty since the hardware is customized to meet the requirements of the algorithm.

This chapter is organized as follows. In Section 2, a brief introduction to pattern recognition systems is given, followed by definition of the terminology used in fingerprint matching, and introduction of various stages in an AFIS. Section 3 briefly reviews the Splash 2 architecture and its programming paradigm. The fingerprint matching algorithm and its computational requirement are briefly presented in Section 4. The hardware-software design is presented in Section 5. The hardware component of the parallel algorithm has been simulated using the Splash simulator. The results of simulation and synthesis are discussed in Section 6. The synthesized logic has been executed on a set of actual fingerprints. For measuring execution speed, a synthetic database of 10,000 fingerprints has been created from 100 real fingerprints. The execution speed of the matching module is analyzed in Section 7, followed by conclusions in Section 8.

10.2 BACKGROUND

This section is devoted to an introduction to pattern recognition systems, some basic definitions with respect to fingerprints, and automatic fingerprint identification systems (AFIS).

10.2.1 Pattern Recognition Systems

Pattern recognition techniques are used to classify or describe complex patterns or objects by means of some measured properties or features. A pattern is an entity, vaguely defined, that could be given a name. A speech waveform, a person's face, and a piston head are examples of patterns. The goals of pattern recognition are to (i) assign a pattern to a heretofore unknown class of patterns (clustering) or (ii) identify a pattern as a member of an already known class (classification). Two examples of the recognition problem are identifying a suspect in a criminal case based on fingerprints, and finding defects in a printed circuit board.

A pattern recognition system (PRS) classifies an object into one of several predefined classes. The input to a PRS is a set of N measurements represented by an N-dimensional vector called a pattern vector. A PRS can be used to completely automate the decision-making process without any human intervention. A PRS requires data acquisition via some sensors, data representation, and data analysis or classification. The data are usually either in the form of pictures, as in the case of fingerprint matching, or one-dimensional time signals, as in the case of speech recognition. Although these images or signals can be interpreted, analyzed, or classified by trained human operators, pattern recognition systems can provide more reliable and faster analysis, often at a lower cost.

The design of a PRS involves the following three steps:

- Sensing
- Representation
- Decision making

The problem domain influences the choice of sensor, representation, and decision making model. An ideal representation is characterized by the following desirable properties; it is

1. Provided with discriminatory information at several levels of resolution (detail)
2. Easily computable
3. Amenable to automated matching algorithms
4. Stable and invariant to noise and distortions
5. Efficient and compact

The compactness property of a representation often constrains its discriminating power.

The pattern recognition problem is difficult because various sources of noise distort the patterns, and often there exists a substantial amount of variability among the patterns belonging to the same category [5]. For example, the character 'A' written by different people looks different, though we assign the same class label 'A' to all of them. Hence, it is not appropriate to use the raw pattern vector for classification. Invariant features that characterize a set of patterns are used to represent a class of patterns. Several issues arise, such as what features should be used and how they should be extracted reliably. The features of a pattern are the input to a classification stage. The challenge in designing a recognition system is in extraction of features that can tolerate the intra-class variations and still possess the inter-class discriminating power. If the extracted features have sufficient discriminating power, then the decision making stage is simple. Conversely, a sophisticated decision making stage can compensate for an unreliable feature extraction stage. In practice, we never have a noiseless input pattern, an ideal representation, perfect feature extraction, or robust decision maker. Imperfections in any of these stages may result in classification error. The goal of a pattern recognition system is to minimize the classification error. Many successful pattern recognition systems have been built in the area of document analysis, medical diagnosis, and fingerprint identification. A large number of books and survey papers have been written on pattern recognition. Readers interested in more details are referred to [5].

10.2.2 Terminology

The structural features that are commonly extracted from the gray-level input fingerprint image are ridge bifurcations and ridge endings. Each of the features has three components, namely, the x-coordinate, the y-coordinate, and the local ridge direction at the feature location, as shown in Figure 10.4. Many other features that have been used for fingerprint matching are derived from this basic three-dimensional feature vector [1].

Definitions of some relevant fingerprint-related terms are given below. Readers interested in more details are referred to [2].

- Fingerprint image: A digitized image of a fingerprint impression usually containing 512×512 pixels and 256 gray levels.
- Fingerprint card: A paper card with a provision to record impressions of all 10 fingers of a person, including other textual details (such as name, sex, and age) useful for identification.

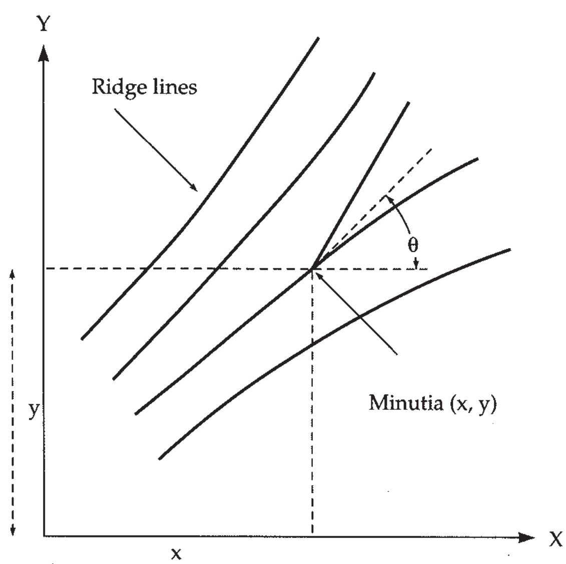


FIGURE 10.4 Components of a Minutia Feature

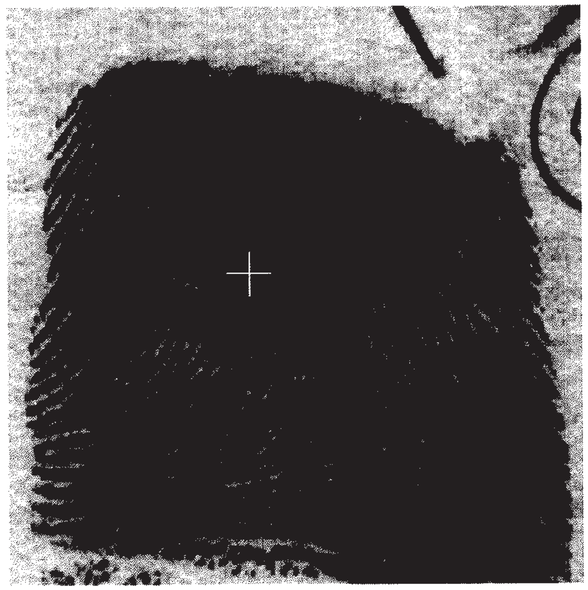


FIGURE 10.5 A Core Point Marked on a Gray-level Fingerprint

- Pattern area: The area of the image where the fingerprint pattern is located.
- Ridge: A black line in a fingerprint image. See Figure 10.1.
- Valley: A white line in a fingerprint image. See Figure 10.1.
- Ridge bifurcation point: A point where a ridge branches into two ridges. See Figure 10.2(a).
- Ridge end point: A point where a ridge stops flowing. See Figure 10.2(b).
- Minutia: A ridge ending or bifurcation point.
- Classification: Based on the ridge flow type, the process of categorizing fingerprints into one of the following five classes: (i) arch, (ii) loop, (iii) whorl, (iv) double loop, and (v) accidental. The first three fingerprint classes are shown in Figure 10.1.
- Matching: The process of comparing a pair of fingerprints based on their minutiae feature sets. The AFIS systems usually determine a list of probables (possible matches) from the database, often sorted on a matching score that indicates the degree of match.
- Core point: For whorls, loops, and double loops, the core point is defined as the topmost point on the innermost ridge, assuming the fingerprint is oriented. See Figure 10.5.

10.2.3 Stages in AFIS

An AFIS is a pattern recognition system for fingerprint matching. A typical AFIS consists of various processing stages as shown in Figure 10.6. For the purpose of automation, a suitable representation of fingerprints is essential. Clearly, the raw digital image (set of pixels) of a fingerprint itself does not meet the requirements of an ideal representation described earlier. Hence, high-level structural features are extracted from the fingerprint image for the purpose of representation and matching.

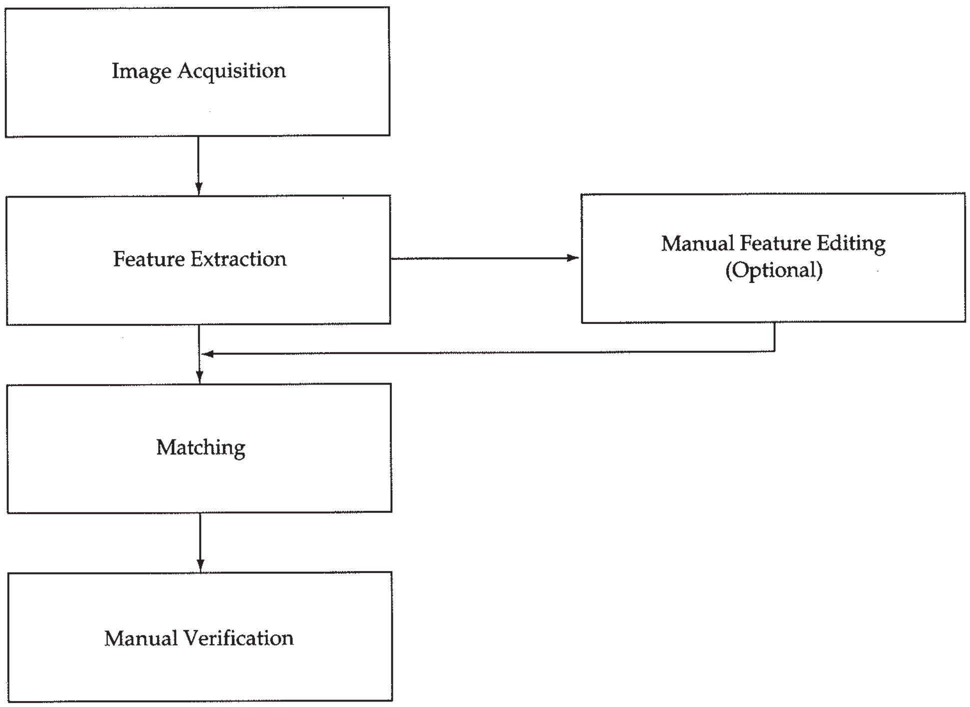


FIGURE 10.6 Stages in an Automatic Fingerprint Identification System (AFIS)

The commercially available fingerprint systems typically use ridge bifurcations and ridge endings as features (see Figure 10.2). Because of the large size of the fingerprint database and the noisy fingerprints encountered in practice, it is very difficult to achieve a reliable one-to-one matching in all test cases. Therefore, the commercial systems provide a ranked list of possible matches (usually the top 10 matches) that are then verified by a human expert. The matching stage uses the position and orientation of these features and the total number of such features. As a result, the accuracy of feature extraction has a strong impact on the overall accuracy of fingerprint matching. Reliable and robust features can simplify the matching algorithm and obviate the manual verification stage.

One of the main problems in extracting structural features is the presence of noise in the fingerprint image. Commonly used methods for taking fingerprint impressions involve applying a uniform layer of ink on the finger and rolling the finger on paper. This leads to the following problems. Smudgy areas in the image are created by overinked areas of the finger, while breaks in ridges are created by underinked areas. Additionally, the elastic nature of the skin can change the positional characteristics of the fingerprint features depending on the pressure applied on the fingers. Though inkless methods for taking fingerprint impressions are now available, these methods still suffer from the positional shifting caused by the skin elasticity. The AFIS used for criminal identification poses yet another problem. A noncooperative attitude of suspects or criminals in providing the impressions leads to smearing parts of the fingerprint impression. Thus, noisy features are inevitable in real fingerprint images.

The matching module must be robust to overcome the noisy features generated by the feature extraction module.

The functioning of an AFIS can be described starting with the input stage. A gray-scale fingerprint image is obtained using a scanner or a camera. Recently, inkless methods have been used for this stage [7]. The input image needs enhancement before further processing can be done. This stage involves image processing techniques to minimize noise and enhance image contrast. A feature extraction stage locates the minutiae points in the enhanced image. Often, it is difficult to extract minutiae reliably from noisy inputs. In such cases, a human fingerprint expert interactively updates the location of the minutiae. The set of minutiae forms the input to a matcher. The matcher reads fingerprint features from the database and matches these with the query fingerprint feature set. It outputs a list of probables from the database in order of their degree of match. The system output is verified by the human expert to arrive at the final decision for each query fingerprint.

10.3 SPLASH 2 ARCHITECTURE AND PROGRAMMING MODELS

We review the major components of the Splash 2 system that are used by our fingerprint matching algorithm. (For details, refer to the chapters on Splash 2 architecture and programming.)

Each Splash 2 processing board has 16 Xilinx 4010s as Processing Elements (PEs X1–X16) in addition to a seventeenth Xilinx 4010 (X0) that controls the data flow into the processor board. Each PE has 512 KB of memory. The Sun SPARCstation host can read/write this memory. The PEs are connected through a crossbar that is programmed by X0. There is a 36-bit linear data path (SIMD Bus) running through all the PEs. The PEs can read data either from their respective memory or from any other PE. A broadcast path also exists by suitably programming X0.

The Splash 2 system supports several models of computation, including PEs executing a single instruction on multiple data (SIMD mode) and PEs executing multiple instructions on multiple data (MIMD mode). It can also execute the same or different instructions on single data by receiving data through the global broadcast bus. The most common mode of operation is systolic, in which the SIMD Bus is used for data transfer. Also, individual memory available with each PE is used to conveniently store temporary results and tables.

To program Splash 2, we need to program each of the PEs (X1–X16), the crossbar, and the host interface. The crossbar sets the communication paths for any arbitrary pattern of communication between PEs. In case the crossbar is used, X0 needs to be programmed. The host interface handles data transfers in and out of the Splash 2 board.

10.4 FINGERPRINT MATCHING ALGORITHM

The feature extraction process takes the input fingerprint gray-level image and extracts the minutiae features described in Section 1, making no efforts to distinguish between the two categories (ridge endings and ridge bifurcations). Figure 10.7 shows

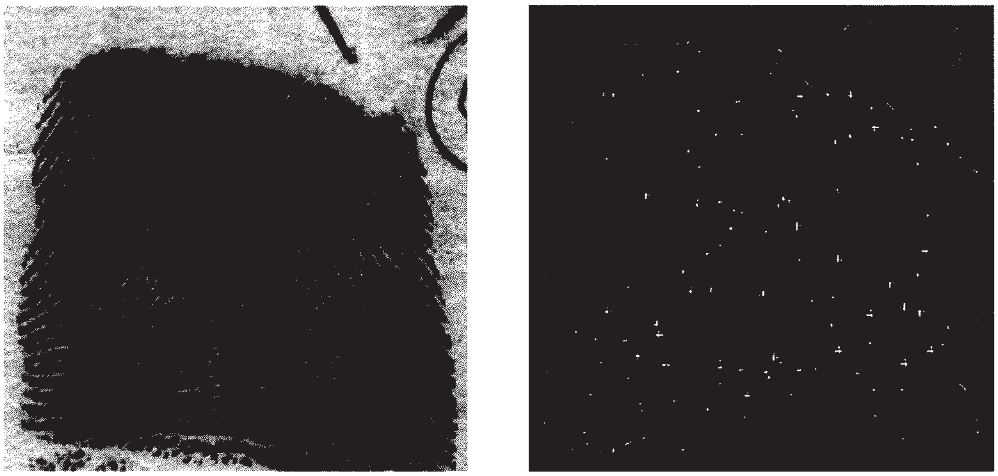


FIGURE 10.7 Feature Extraction. (a) A gray-scale image of a fingerprint; (b) its skeleton with features

a gray-scale fingerprint image and its skeleton image where these features are marked. In this section, an algorithm for matching rolled fingerprints against a database of rolled fingerprints is presented. A query fingerprint is matched with every fingerprint in the database, discarding candidates whose matching scores are below a user-specified threshold. Rolled fingerprints usually contain a large number of minutiae (between 50 and 100). Since the main focus of this chapter is on parallelizing the matching algorithm, we assume that the features have been extracted from the fingerprint images and the important information is available. In particular, we assume that the core point of the fingerprint is known and that the fingerprints are oriented properly.

10.4.1 Minutia Matching

Matching a query and database fingerprint is equivalent to matching their minutiae sets. Each query fingerprint minutia is examined to determine whether there is a corresponding database fingerprint minutia. Two minutiae are said to be *paired* or *matched* if their components (x, y, θ) are equal within some tolerance after registration, which is the process of aligning the two sets of minutiae along a common core point (see section 4.2 for precise definitions). Three situations arise as shown in Figure 10.8.

1. A database fingerprint minutia matches the query fingerprint minutia in all the components (paired minutiae);
2. A database fingerprint minutia matches the query fingerprint minutia in the x and y coordinates, but does not match in the direction (minutiae with unmatched angle);
3. No database fingerprint minutia matches the query fingerprint minutia (unmatched minutia).

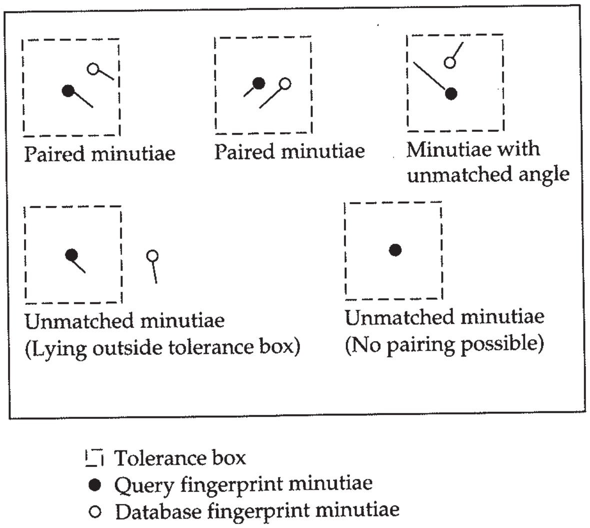


FIGURE 10.8 Different Situations in Minutia Matching

Of the three cases described above, only in the first case are the minutiae said to be paired.

10.4.2 Matching Algorithm

The following notation is used in the sequential and parallel algorithms described below. Let the query fingerprint be represented as an n-dimensional feature vector $\mathbf{f^q} = (\mathbf{f^q_1}, \mathbf{f^q_2}, ..., \mathbf{f^q_n})$. Note that each of the n elements is a feature vector corresponding to one minutia, and the ith feature vector contains three components, $\mathbf{f_i} = (f_i(x), f_i(y), f_i(\theta))$.

The components of a feature vector are shown geometrically in Figure 10.4. The query fingerprint core point is located at (C^q_x, C^q_y). Similarly, let the rth reference (database) fingerprint be represented as an m_r-dimensional feature vector $\mathbf{f^r} = (\mathbf{f^r_1}, \mathbf{f^r_2}, ..., \mathbf{f^r_{m_r}})$, and the reference fingerprint core point is located at (C^r_x, C^r_y).

Let (x^t_q, y^t_q) and (x^b_q, y^b_q) define the bounding box for the query fingerprint, where x^t_q is the x-coordinate of the top-left corner of the box and x^b_q is the x-coordinate of the bottom-right corner of the box. Quantities y^t_q and y^b_q are defined similarly. A bounding box is the smallest rectangle that encloses all the feature points. Note that the query fingerprint $\mathbf{f^q}$ may or may not belong to the fingerprint database $\mathbf{f^D}$. The fingerprints are assumed to be registered with a known orientation. Hence, there is no need of normalization for rotation.

The matching algorithm is based on finding the number of paired minutiae between each database fingerprint and the query fingerprint. It uses the concept of minutiae matching described in Section 4.1. A tolerance box is shown graphically in Figure 10.9. In order to reduce the amount of computation, the matching algorithm takes into account only those minutiae that fall within a common bounding box.

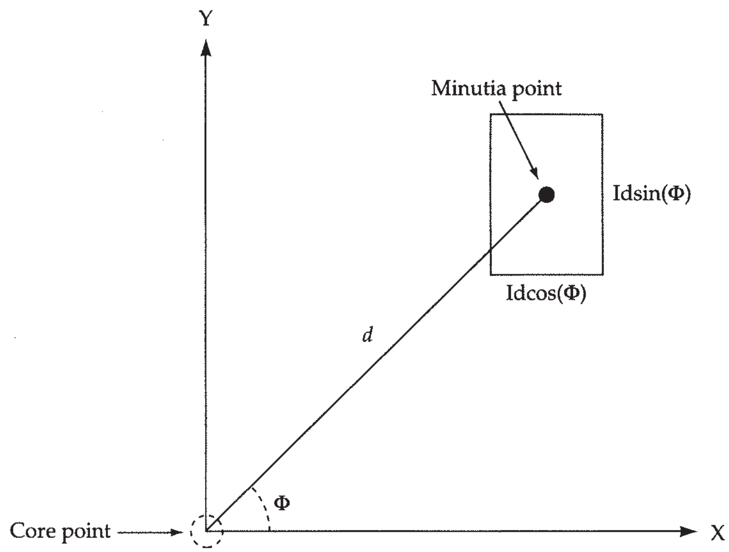


FIGURE 10.9 Tolerance Box for X- and Y-components of a Minutia Point

The common bounding box is the intersection of the bounding box for query and reference (database) fingerprints. Once the count of matching minutiae is obtained, a matching score is computed. The matching score is used for deciding the degree of match. Finally, a set of top-scoring reference fingerprints is obtained as a result of matching.

In order to accommodate the shift in the minutia features, a tolerance box is created around each feature. The size of the box depends on the ridge widths and distance from the core point in the fingerprint.

The sequential matching algorithm is described in Figure 10.10. In the sequential algorithm, the tolerance box (shown in Figure 10.9 with respect to a query fingerprint minutia) is calculated for the reference (database) fingerprint minutia. In the parallel algorithm described in the next section, it is calculated for the query fingerprint (as in Figure 10.9). A similar sequential matching algorithm is described by Wegstein [9]. Depending on the desired accuracy, more than one finger could be used in matching. In that case, a composite score is computed for each set.

10.5 PARALLEL MATCHING ALGORITHM

We parallelize the matching algorithm so that it utilizes the specific characteristics of the Splash 2 architecture. While performing this mapping, we need to take into account the limitations of the available FPGA technology. This is consistent with the approaches taken in hardware-software codesign. Any preprocessing needed on the query minutiae set is a one-time operation, whereas reference fingerprint minutiae matching is a repetitive operation. Computing the matching score involves floating-point division. The floating-point operations and one-time operations are performed in software on the host, whereas the repetitive operations are delegated to the FPGA-based PEs of Splash 2. The parallel version of the algorithm involves operations on

Input: Query fingerprint n-dimensional feature vector $\mathbf{f^q}$ and the rolled fingerprint database $\mathbf{f^D} = \{\mathbf{f^r}\}_{r=1}^{N}$.
The rth database fingerprint is represented as an m_r-dimensional featurevector.
Output: A list of top ten records from the database with matching score > T.
Begin

For $r = 1$ to N do

1. Register the database fingerprint with respect to the core point (C_x^q, C_y^q) of the query fingerprint:

For $i = 1$ to m_r do
$f_i^r(x) = f_i^r(x) - C_x^q$
$f_i^r(y) = f_i^r(y) - C_y^q$

2. Compute the common bounding box for the query and reference fingerprints:
Let (x_q^t, y_q^t) and (x_q^b, y_q^b) define the bounding box for the query fingerprint.
Let (x_r^t, y_r^t) and (x_r^b, y_r^b) define the bounding box for the rth reference fingerprint.
The intersection of these two boxes is the common bounding box.
Let the query print have M_e^q and reference print have N_e^r minutiae in this box.

3. Compute the tolerance vector for ith feature vector f_i^r:
If the distance from the reference core point to the current reference feature is less than K then
$t_i^r(x) = ld\cos(\phi)$,
$t_i^r(y) = ld\sin(\phi)$, and
$t_i^r(\emptyset) = k_3$,
else
$t_i^r(x) = k_1$,
$t_i^r(y) = k_2$, and
$t_i^r(\emptyset) = k_3$,
where l, k_1, k_2 and k_3 are prespecified constants determined empirically based on the average ridge width,
ϕ is the angle of the line joining the core point and the ith feature with the x-axis,
and d is the distance of the feature from the core point.
Tolerance box is shown geometrically in Figure 10.9.

4. Match minutiae:
Two minutiae $\mathbf{f_i^r}$ and $\mathbf{f_j^q}$ are said to match if the following conditions are satisfied:
$f_j^q(x) - t_i^r(x) \leq f_i^r(x) \leq f_j^q(x) + t_i^r(x)$,
$f_j^q(y) - t_i^r(y) \leq f_i^r(y) \leq f_j^q(y) + t_i^r(y)$, and
$f_j^q - t_i^r(\theta) \leq f_i^r(\theta) \leq f_j^q(\theta) + t_i^r(\theta)$,
where $t_i^r = (t_i^r(x), t_i^r(y), t_i^r(\theta))$ is the tolerance vector.
Set the number of paired features, $m_p^r = 0$;
For all query features $\mathbf{f_j^q}$, $j = 1,2, \ldots M_e^q$, do
If $\mathbf{f_j^q}$ matches with any feature in $\mathbf{f_i^r}$, $i = 1,2, \ldots, N_e^r$, then increment m_p^r.
Mark the corresponding feature in $\mathbf{f^r}$ as paired.

5. Compute the matching score (MS (q,r)):
$MS(q, r) = \frac{m_p^r * m_p^r}{(M_e^q * N_e^r)}$.

Sort the database fingerprints and obtain top 10 scoring database fingerprints.

End

FIGURE 10.10 Sequential Fingerprint Matching Algorithm

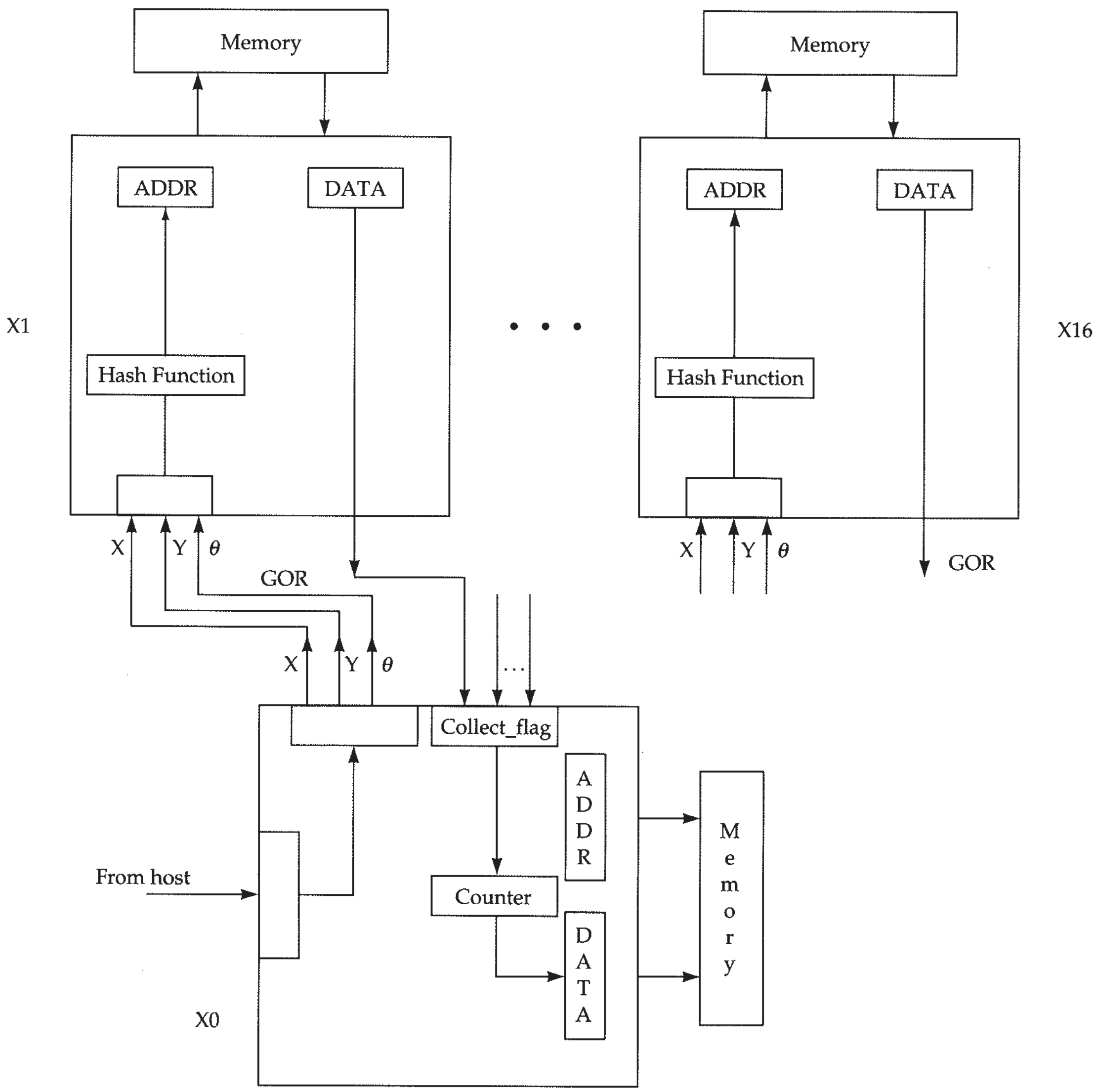


FIGURE 10.11 Fingerprint Matching in Splash 2

the host, on X0, and on each PE. The schematic of fingerprint matching algorithm using Splash 2 is shown in Figure 10.11.

One of the main constructs of the parallel algorithm is a lookup table. The lookup table consists of all possible points within the tolerance box that a feature may be mapped to. The Splash 2 data paths for the parallel algorithm are shown in Figure 10.12.

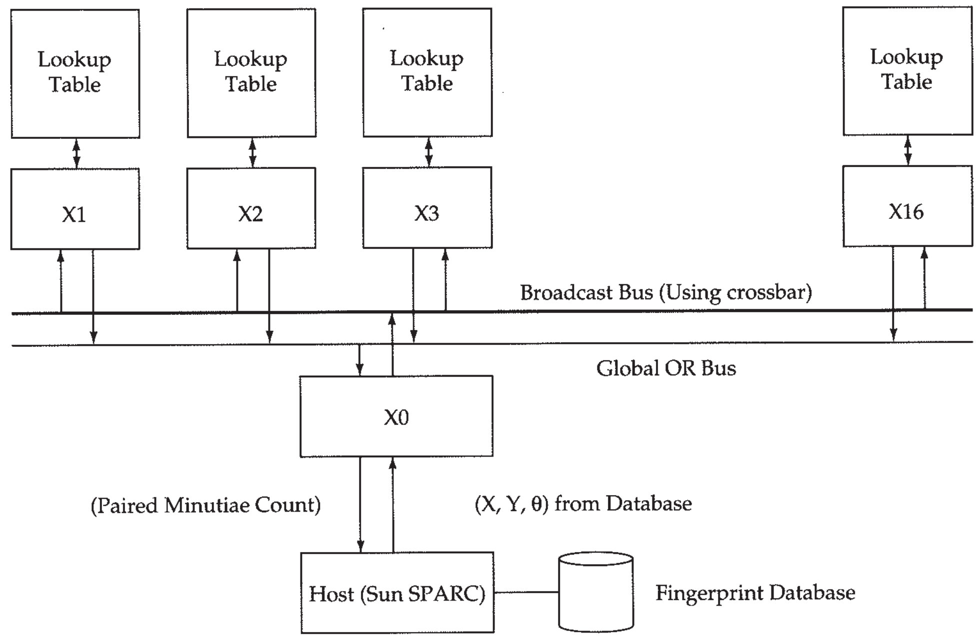


FIGURE 10.12 Data Flow in Parallel Matching Algorithm

10.5.1 Preprocessing on the Host

The host processes the query and database fingerprints as follows. The query fingerprint is read first and the following preprocessing is done:

1. The core point is assumed to be available. For each query feature $\mathbf{f}_j^q$, $j = 1, 2, \ldots n$, generate a tolerance box. Enumerate a total of $(t_x \times t_y \times t_\theta)$ grid points in this box, where t_x is the tolerance in x, t_y is the tolerance in y and t_θ is tolerance in θ.
2. Allocate each feature to one PE in Splash 2. Repeat this cyclically, that is, features 1–16 are allocated to PEs X1 to X16, features 17–32 are allocated to PEs X1 to X16, and so on.
3. Initialize the lookup tables by loading the grid points within each tolerance box in step (1) into the memory.

In this algorithm, the tolerance box is computed with respect to the query fingerprint features. The host then reads the database of fingerprints and sends their feature vectors for matching to the Splash 2 board.

For each database fingerprint, the host performs the following operations:

1. Read the feature vectors.
2. Register the features as described in step (1) of the sequential algorithm in Figure 10.10.
3. Send each of the feature vectors over the broadcast bus to all PEs if it is within the bounding box of the query fingerprint.

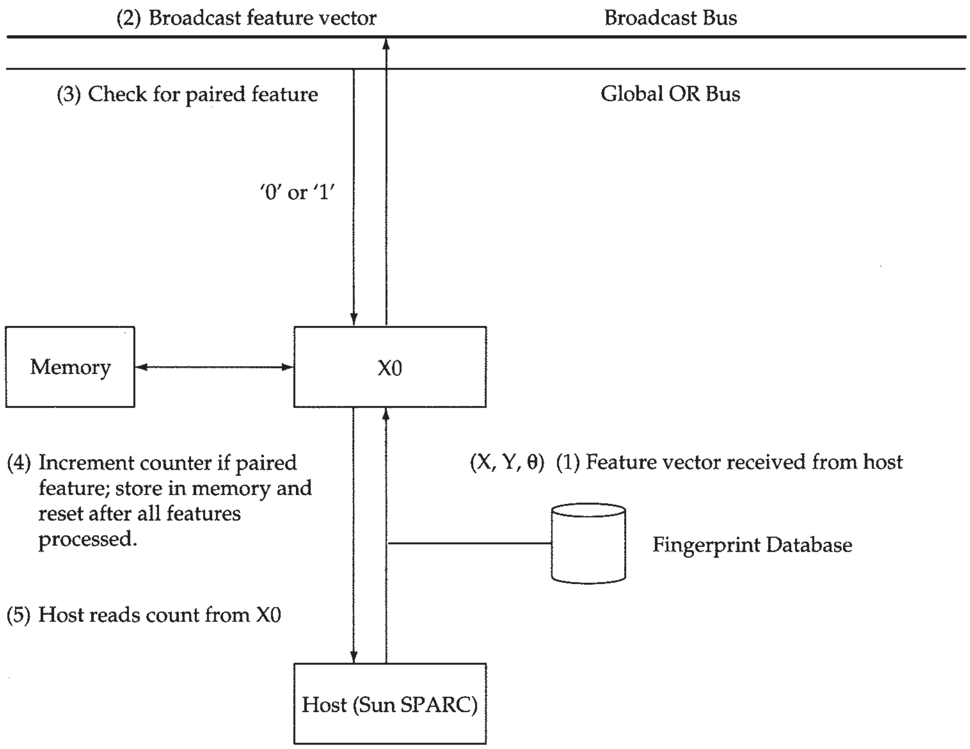


FIGURE 10.13 Data Flow in X0

For each database fingerprint, the host then reads the number of paired features m_p^r that was computed by the Splash 2 system, $r = 1, \ldots N$. Finally, the matching score is computed as in the sequential method.

10.5.2 Computations on Splash

The computations carried out on each PE of Splash 2 are described below. As mentioned earlier, X0 plays a special role in controlling the crossbar in Splash 2.

1. Operations on X0:
   Each database feature vector received from the host is broadcast to all PEs. If it is matched with a feature in a lookup table, the PE drives the Global OR bus high. When the OR bus is high, X0 increments a counter. The host reads this counter value (m_p^r) after all the feature vectors for the current database fingerprint have been processed. Operations on X0 are highlighted in Figure 10.13.
2. Operations on each PE:
   On receiving the broadcasted feature, a PE computes its address in the lookup table through a hashing function. If the data at the computed address is a '1', then the feature is paired, and the PE drives the Global OR bus high. Operations on a PE are highlighted in Figure 10.14.

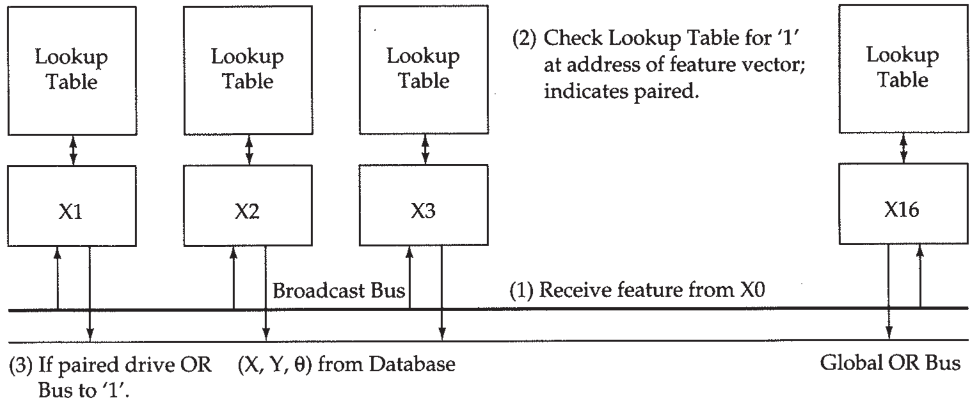


FIGURE 10.14 Data Flow in a PE

10.5.3 VHDL Specification for X0

We illustrate how the operations on X0 are customized by describing segments of its VHDL code. The tasks carried out by the other PEs are relatively simpler. The following functions are carried out by X0:

1. Broadcast feature vector to all PEs
2. Update a counter if at least one of the bits of the Global OR bus is '1', and
3. Reset the counter after all the minutiae of a database fingerprint are processed and the result is updated in X0 memory.

Five segments of VHDL code are shown in Figure 10.15 and are briefly described here. Segment 1 (lines 1.1–1.7) shows the signal declarations. The hardware buses have been directly mapped to bit vectors in VHDL. Some of the program variables have been tailored for the range needed based on the application requirement (such as *count, features*). Segment 2 describes the padding instructions. Note that because of using input-output pads, there is a delay in a signal reaching all the PEs after it has been seen by X0. The delay is accounted for by using a data pipeline of suitable length (in our case the pipeline is 6 stages deep). The code in line 1.7 combined with code segment 5 (line 5.1) show the use of the pipeline. X0 maintains this pipeline by writing data into the pipeline and flushing out the last data sets by writing zeros. The code in X0 looks at the end of the pipeline. Thus, the data is seen by X0 code when it would have reached other PEs.

By setting suitable configuration parameters, X0 can be set to broadcast the contents of the SIMD Bus to all PEs. To set this mode, code segment 3 is used.

In code segment 4, the collection of OR flags from all 16 PEs (PE X1 through X16) is being checked for any possible match by comparing with a bit vector of all 0's. If any of the bits is a '1', we increment the counter *count*.

If the input for a new database record is initiated, indicated by the 33rd bit of the SIMD bus, then the final paired count and the number of features for the previous record is stored in memory. The two counters *count* and *features* are reset to zero. These activities are carried out in code segment 5.

```
—     Signal declarations — (Segment 1)

1.1-  SIGNAL   Data          :     Bit_Vector(15 downto 0);
1.2-  SIGNAL   Address       :     Bit_Vector(17 downto 0);
1.3-  SIGNAL   count         :     natural range 0 to 255 := 0;
1.4-  SIGNAL   features      :     natural range 0 to 255 := 0;
1.5-  SIGNAL   SIMD          :     Bit_Vector(35 downto 0);
1.6-  SIGNAL   Collect_flag  :     Bit_Vector(15 downto 0);
1.7-  SIGNAL feat_pipeline   pipeline;

—     Connections to I/O pads — (Segment 2)

2.1-  pad_output  (X0_Mem_A, Address);
2.2-  pad_output  (X0_Mem_D, Data);
2.3-  pad_Input   (X0_SIMD, SIMD);
2.4-  pad_Output  (X0_XB_Data, Xbar_Out);
2.5-  pad_Input   (X0_GOR_Result_in, Collect_flag);

—     Setting X0 to be the crossbar master — (Segment 3)

3.1-  X0_Xbar_En_L <= '0';
3.2-  X0_X16_Disable <= '1';
3.3-  X0_Xbar_Send <= '1';

— ..... — (Segment 4)

4.1-  IF (Collect_flag /= itobv(0,16)) THEN
4.2-           count <= count + 1;
4.3-  END IF;

— .....
— New person record, store present counters and then reset — (Segment 5)

5.1-  IF (feat_pipeline(0)(32) = '1') THEN
5.2-           Data(7 downto 0) <= itobv(count,8);
5.3-           Data(15 downto 8) <= itobv(features,8);
5.4-           count <= 0;
5.5-           features <= 0;
5.6-           Address <= itobv(bvtoi(Address) + 1,18);
5.7-  END IF;
```

FIGURE 10.15 VHDL Specification Segments for X0

10.6 SIMULATION AND SYNTHESIS RESULTS

The VHDL behavioral modeling code for PEs X0–X16 has been tested using the Splash simulation environment. The simulation environment loads the lookup tables and crossbar configuration file into the simulator. Note that the Splash simulator runs independently of the Splash 2 hardware and runs on the host. The input data are read from a specified file, and the data on each of the signals declared in the VHDL code can be traced as a function of time. A sample output of simulation using test inputs

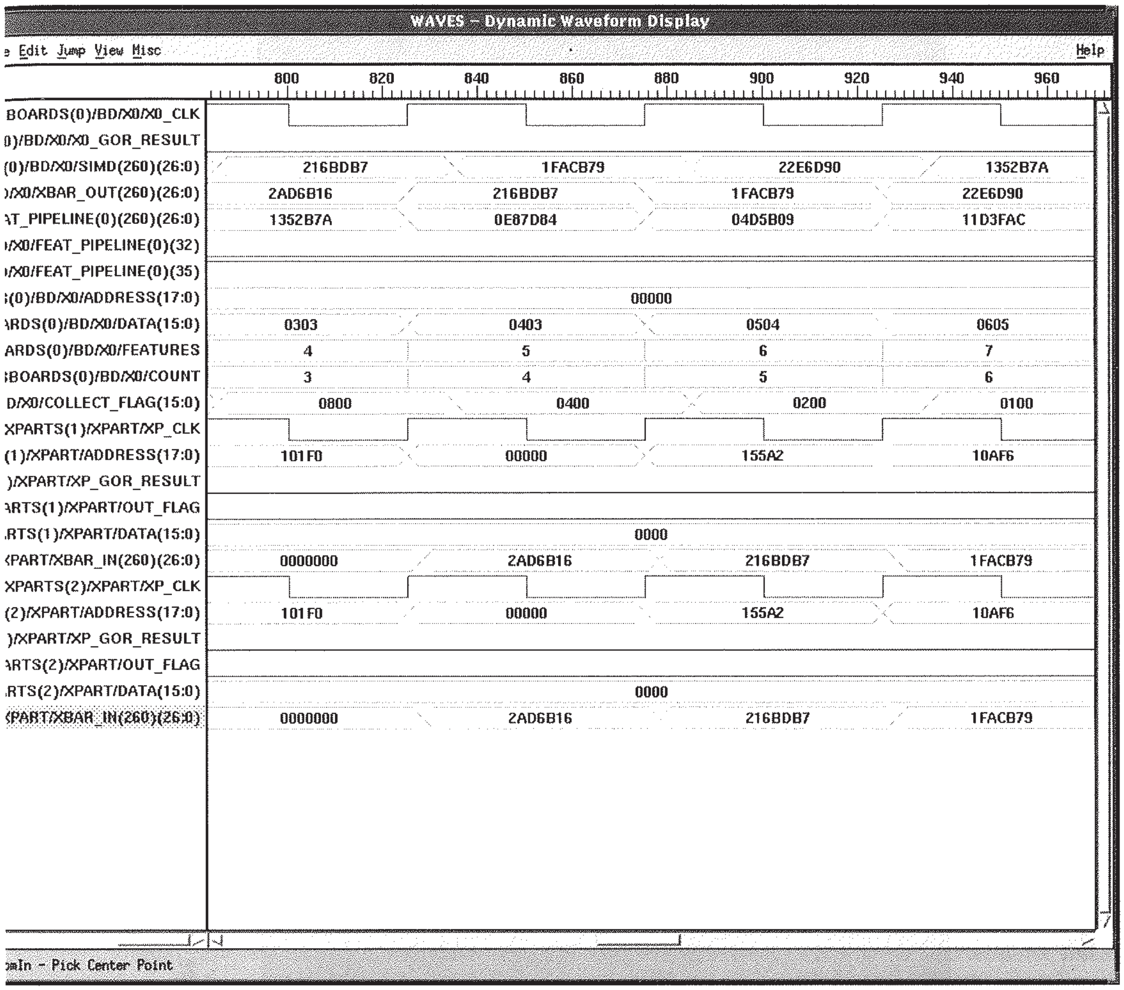
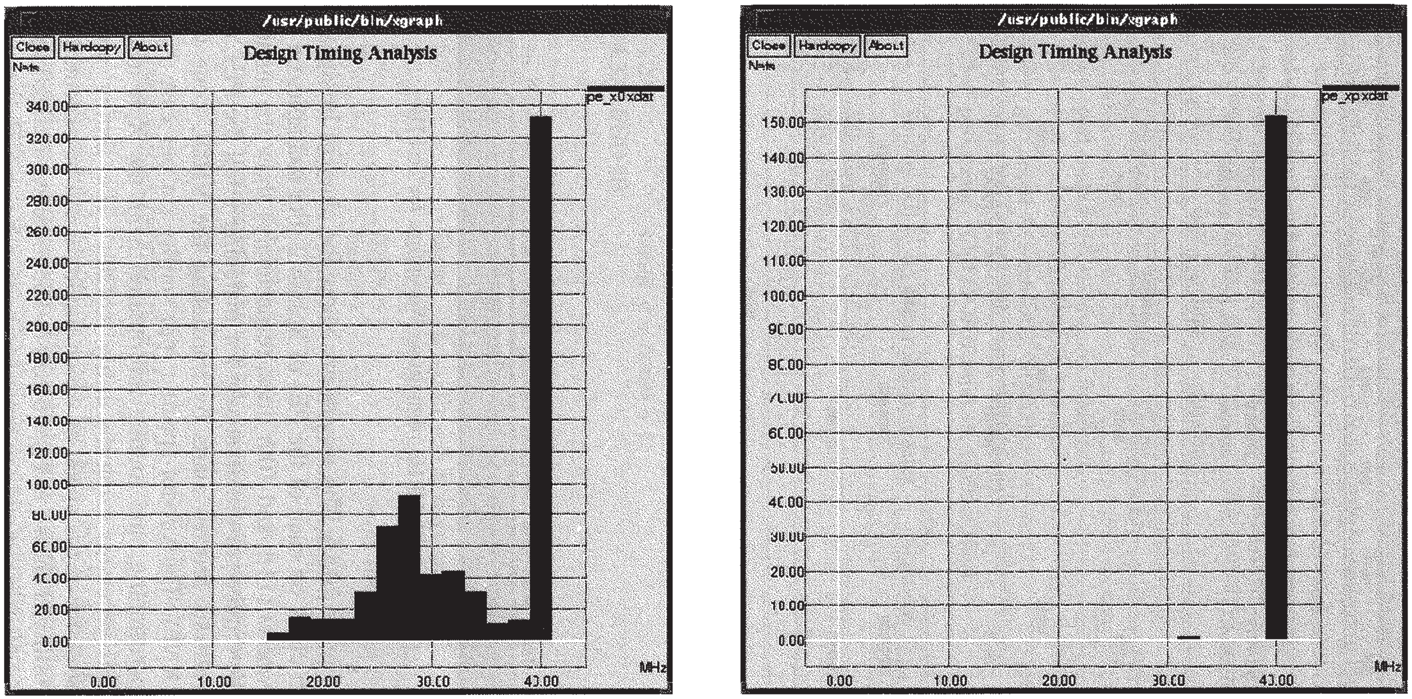


FIGURE 10.16 Simulation Waveforms for Test Data

is shown in Figure 10.16. The waveforms show the changes in signals with respect to the system clock on each of the PEs of Splash 2. For example, on X0, the signals *count* and *features* (11th and 10th lines, respectively) show the number of minutiae paired and the number of minutiae sent for matching to all the PEs, respectively.

The synthesis process starts by translating the VHDL code to a Xilinx net list format (XNF). The vendor-specific 'ppr' utility (in our case Xilinx) generates placement, partitioning, and routing information from the XNF net list. The final bitstream file is generated using the utility 'xnf2bit'. The 'timing' utility produces a graphical histogram of the speed at which the logic can be executed. The output of the 'timing' utility is shown in Figure 10.17. The logic synthesized for X0 can run at a clock rate of 17.1 MHz, and the logic for the PEs X1 to X16 can run at 33.8 MHz. Observe that these clock rates correspond to the longest delay (critical) paths, even though most of the logic could be driven at higher rates. Increased processing speed may be possible by optimizing the critical path.



(a) (b)

FIGURE 10.17 Timing Results. (a) for PE X0; (b) for PE X_i

10.7 EXECUTION ON SPLASH 2

The bitstream files for Splash 2 are generated from the VHDL code. Using the C interface for Splash 2, a host version of the fingerprint matching application is generated. The host version reads the fingerprint database from the disk and obtains the final list of candidates after matching.

10.7.1 User Interface

An interactive user interface to the fingerprint matching application has been developed using the X Window System. The interface provides pull-down menus for selecting a query fingerprint for matching and for invoking tasks of feature extraction, matching, and verification. The graphical user interface is shown in Figure 10.18. The matching menu can select either the host or Splash 2 to perform the computations during matching. The speed of matching is computed by obtaining the elapsed time for the number of fingerprints in the database.

10.7.2 Performance Analysis

The sequential algorithm, described in Section 4.2, executed on a Sun SPARCstation 10, performs at the rate of 70 matches per second on database and query fingerprints that have approximately 65 features. A match is the process of determining the matching score between a query and a reference fingerprint. The Splash 2 implementation should perform matching at the rate of 2.6×10^5 matches per second. This matching speed is obtained from the 'timing' utility. The host interface part can run at 17.1 MHz and each PE can run at 33.8 MHz (as shown in Figure 10.17). Hence, the entire fingerprint matching will run at the slower of the two speeds, that is, 17.1 MHz. Assuming 65 minutiae, on an average, in a database fingerprint, the matching speed is estimated at 2.6×10^5 matches per second. We evaluated the matching speed using a database of 10,000 fingerprints created from 100 real fingerprints by randomly dropping, adding, and perturbing minutiae in a given set of minutiae. The measured speed on a Splash 2 system running at 1 MHz is of the order of 6,300 matches per second on this database. The experimental Splash 2 system has not been run at higher clock rates. Assuming a linear scaling of performance with an increase in clock rate, we would achieve approximately 110,000 matches per second. We feel that the disparity in the projected and achieved speeds (2.6×10^5 versus 1.1×10^5) is due to different tasks being timed. The time to load the data buffers onto Splash 2 has not been taken into account in the projected speed, whereas this time is included in the time measured by the host in an actual run. We are in the process of timing only the matching component of the code on the system.

The main advantage of the Splash 2 implementation is the higher performance compared to the sequential implementation. The Splash 2 implementation is over 1,500 times faster than a sequential implementation on a SPARCstation 10. Another advantage of the parallel implementation on Splash 2 is that the matching speed is independent of the number of minutiae in the query fingerprint. The number of minutiae affects only the lookup table initialization, which is done as preprocessing by the host, and this time is amortized over a large number of database records.

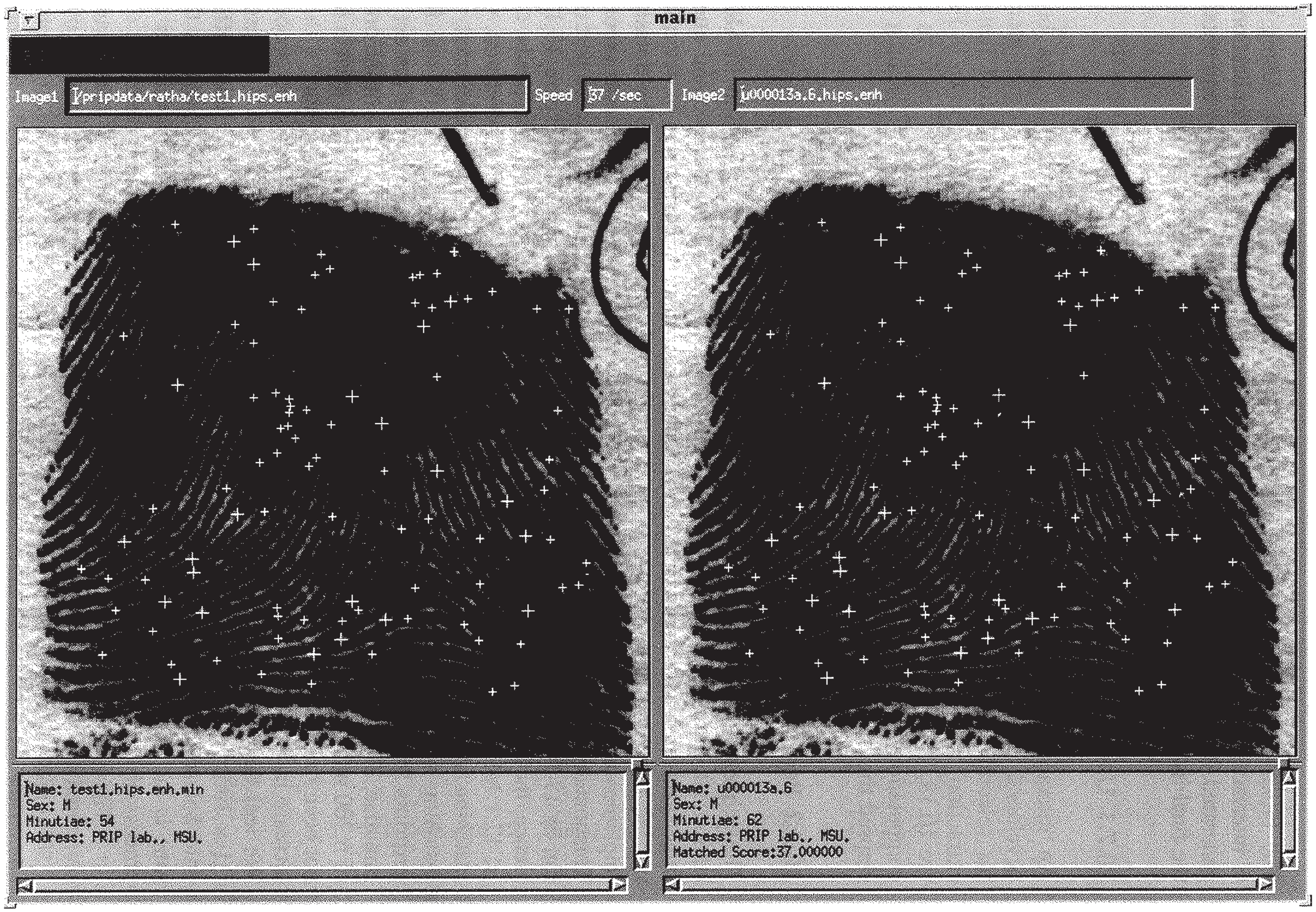


FIGURE 10.18 GUI Used in Fingerprint Analysis

The matching algorithm can scale well as the number of Splash 2 boards on the system is increased. Multiple query fingerprints can be loaded on different Splash 2 boards, each matching against the database records as they are transferred from the host. This would result in a higher throughput from the system.

The processing speed can be further improved by replacing some of the soft macros on the host interface part (X0) by hard macros. To sustain the matching rate, the data throughput should be at a rate of over 250,000 fingerprint records per second (with an average of 65 minutiae per record). This may be a bottleneck for the I/O subsystem.

10.8 CONCLUSIONS

The Splash 2 architecture is highly suitable for rolled fingerprint matching. The parallel algorithm has been designed to match the Splash 2 architecture, thereby resulting in substantially better performance. The algorithm applies a hardware-software design approach to maximize the performance of the overall system.

We will be coding our matching algorithm in dbC to evaluate the performance of such a high-level language to express low-level parallelism. This effort will also enable us to compare the development time needed to program Splash 2 using VHDL versus dbC. In the next phase of the project, we plan to implement a minutiae extraction algorithm and a latent fingerprint matching algorithm on Splash 2. Both of these algorithms appear promising for achieving performance gains on the Splash 2 architecture. The minutiae extraction process involves two-dimensional convolution, which has been successfully implemented on Splash 2 [8].

ACKNOWLEDGMENT

We would like to thank Duncan Buell, Jeff Arnold, and Brian Schott of Supercomputing Research Center, Bowie, Maryland, for their help and suggestions. This research was supported by a research contract from the Institute for Defense Analyses, Alexandria, Virginia.

REFERENCES

[1] "Application Briefs: Computer Graphics in the Detective Business," *IEEE Computer Graphics and Applications*, Apr. 1985, pp. 14–17.

[2] Federal Bureau of Investigation, *The Science of Fingerprints: Classification and Uses*, U.S. Govt. Printing Office, Washington, D.C., 1984.

[3] J.H. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kauffmann Publishers, San Mateo, Calif., 1990.

[4] Sir W.J. Herschel, *The Origin of Fingerprinting*, AMS Press, New York, 1974.

[5] A.K. Jain, "Advances in Statistical Pattern Recognition," in *Pattern Recognition Theory and Applications*, P.A. Devijer and J. Kittler, eds., Springer-Verlag, New York, 1987, pp. 1–19.

[6] H.C. Lee and R.E. Gaensslen, *Advances in Fingerprint Technology*, Elsevier, New York, 1991.

[7] B. Miller, "Vital Signs of Identity," *IEEE Spectrum*, Vol. 31, No. 2, Feb. 1994, pp. 22–30.

[8] N.K. Ratha, A.K. Jain, and D.T. Rover, "Convolution on Splash 2," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 204–213.

[9] J.H. Wegstein, *An Automated Fingerprint Identification System*, Special Publication 500–89, Nat'l Bureau of Standards, Washington, D.C., 1982.

CHAPTER 11

High-Speed Image Processing with Splash 2

Peter M. Athanas and A. Lynn Abbott

11.1 INTRODUCTION

Image processing is the problem of extracting useful information from an image or from a sequence of images. Although images can be produced by many different sources (including x-ray sensors, tomographic scanners, acoustic imagers, and computer-graphics programs), the video camera is of particular interest because it generates images that are easily interpreted by a human observer. Unfortunately, the amount of data that is present in a single image is very large, and the methods that are used in biological vision are not well understood. The challenge of image-processing research is therefore to develop computational approaches—both algorithms and hardware—that can accept images and produce useful results at high speed.

Conventional von Neumann machines are commonly used for image processing tasks, but their performance does not begin to approach real-time rates. The usual alternative is to employ special-purpose architectures that have been designed specifically for image processing. These systems can perform at sufficiently high speeds, but at the expense of flexibility; they can perform *only* the tasks that they have been designed to do. Splash 2 represents a third alternative. Custom computing platforms such as Splash 2 are sufficiently flexible that new algorithms can be implemented on existing hardware, and are fast enough that real-time or near-real-time operation is possible.

This chapter describes a real-time image processing system that is based on the Splash 2 general-purpose custom computing platform. Even though Splash 2 was not designed specifically for image processing, this platform possesses architectural properties that make it well suited for the computation and data transfer rates that are characteristic of this class of problems. Furthermore, the price/performance of this system makes it a competitive alternative to conventional real-time image processing systems.

Other important factors for using Splash 2 are prototyping and design verification. The typical hardware design process requires extensive behavioral testing of a new concept before proceeding with a hardware implementation. For any image processing task of reasonable complexity, simulation of a VHDL model with a representative data set on a workstation is prohibitive because of the enormous simulation time required. Days, or even weeks, of processing time are commonly needed to simulate the processing of a single image. Because of this, the designer is often forced into a trade-off as to how much testing can be afforded versus an acceptable risk of allowing an iteration in silicon. The Splash 2 approach permits an automated (or near-automated) transformation of a structural or behavioral VHDL representation into a real-time hardware implementation. The Splash 2 platform can therefore serve not only as a means to evaluate the performance of an experimental algorithm/architecture, but also as a working component in the development and testing of a much larger system.

The next section describes VTSPLASH, a laboratory system based on Splash 2 that has been developed at Virginia Tech [4]. Section 11.3 presents an overview of image-processing fundamentals, and discusses architectural considerations for high-speed operation. Sections 11.4 and 11.5 present two case studies in the development of image processing tasks: a median filter, and Laplacian pyramid generation. Section 11.6 discusses performance issues. Finally, Section 11.7 summarizes the chapter.

11.2 THE VTSPLASH SYSTEM

The adaptive nature of the Splash 2 architecture makes it well suited for the computational demands of image processing. In addition, Splash 2 features a flexible interface design that facilitates customized I/O for situations that cannot be accommodated by the host workstation. A real-time image processing custom computing system (referred to as VTSPLASH) has been constructed based on Splash 2; this is depicted in Figure 11.1.

A video camera or a VCR is used to create a standard RS-170 video stream. The signal produced from the camera is digitized with a custom-built frame grabber card. This board not only captures images, but also performs any needed sequencing or simple pixel operations before the data are presented to Splash 2. The frame grabber card was built with a parallel interface that can be connected directly to the input data stream of the Splash 2 processor. Two processor Array Boards are used in the VTSPLASH laboratory system. The output data produced by Splash 2, which may be a real-time video data stream, image overlay data, or some other form of information, is first presented to another custom board for converting the data to an appropriate format (if necessary). Once formatted, the data are then presented to a commercial image acquisition/display card, which presents the images to a color video monitor. A Sun SPARCstation serves as the Splash 2 host, and is responsible for configuring the Splash 2 arrays and sending runtime commands intermixed with the video stream if needed. The laboratory system can be rapidly reconfigured from one task to another in just a few seconds.

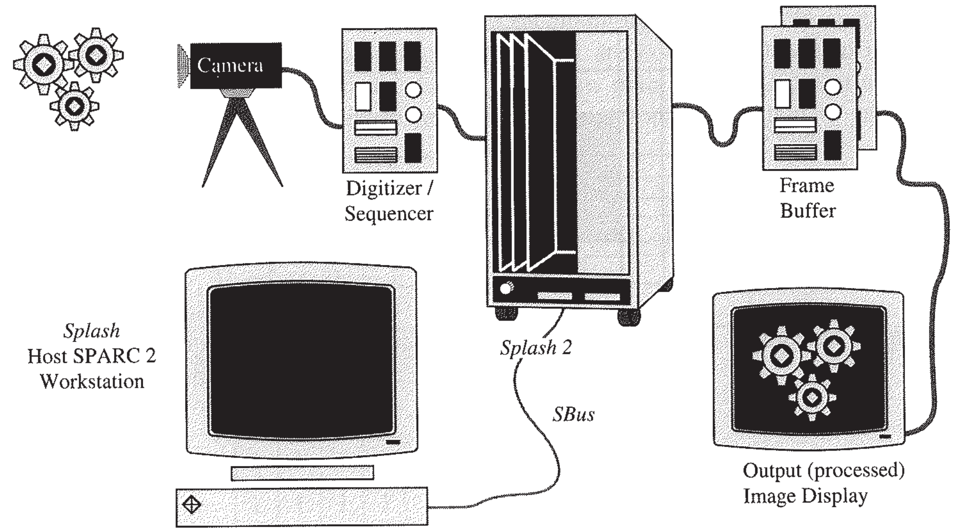


FIGURE 11.1 Components in the VTSPLASH Laboratory System

Although Splash 2 was not specifically designed for image processing, it is a suitable testbed for implementing a wide range of image processing tasks, including those requiring temporal processing. A single Splash 2 processor Array Board contains slightly more than 69 megabits[1] of memory—enough for 32 frames of image data [27]. Not all of this storage is necessarily available to applications in a convenient form; the actual available storage is dependent upon how individual applications are constructed.

11.3 IMAGE PROCESSING TERMINOLOGY AND ARCHITECTURAL ISSUES

A digitized image can be represented as a rectangular array $I(r, c)$, where r and c refer to the row and column location of a picture element, or *pixel*, in the image. For a standard monochrome (black and white) video camera, common image sizes are 512×512 and 480×640 pixels (rows $\times$ columns), where each pixel is an 8-bit quantity representing the light intensity at one point. Since the standard video rate is 30 images per second, even simple tasks represent a significant computational challenge because of the sheer quantity of data: 7.5 MB/s for images of size 512×512. Storage and I/O are also especially significant when real-time operation is required.

The goal of many image processing tasks is to produce an output image I_{out} that is an enhanced or filtered version of an input image I_{in}. One way to accomplish this is to apply a linear filter, $I_{out}(r, c) = \sum_i \sum_j I_{in}(r+i, c+j) \cdot h(i, j)$, where h is the filter and where the summations are performed over a neighborhood determined

[1]This number is based upon seventeen 256K (16 static RAM devices plus 12,800 bits of storage (maximum) in each of the seventeen Xilinx 4010 chips.

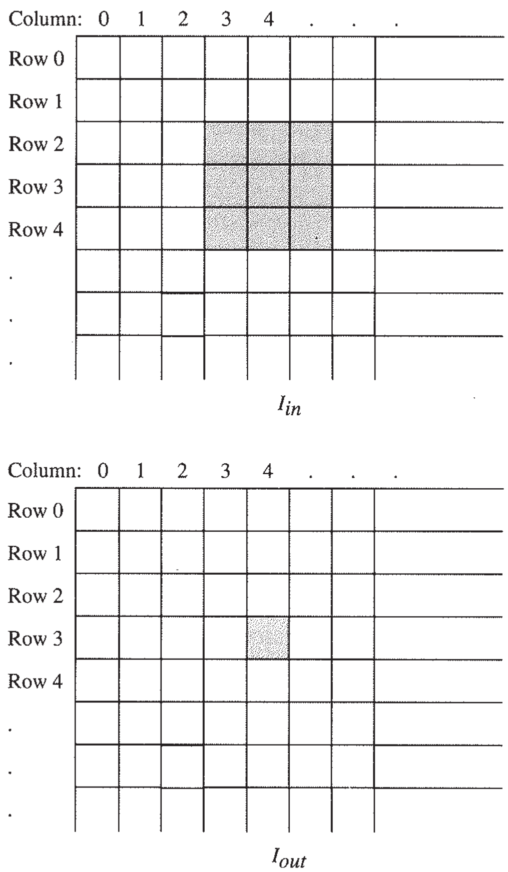


FIGURE 11.2 Example Image Arrays. Each cell represents one pixel, which is commonly 8 bits for a monochrome image. The shaded area at the top indicates a 3 × 3 neighborhood centered about pixel (3, 4). The result of the neighborhood operation is placed in the shaded location at the bottom.

by the extent of h. For example, a smoothed image I_{out} is produced if we define

$$h(i, j) = \begin{cases} \frac{1}{9} \text{ for } -1 \leq i \leq 1 \text{ and } -1 \leq j \leq 1 \\ 0 \qquad\qquad \text{otherwise} \end{cases}.$$

This is equivalent to averaging the pixels within a 3 × 3 neighborhood of I_{in} to produce a single output pixel of I_{out}. This same low-pass filter can be represented as follows:

$$h = 1/9 \times \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

Conceptually, this template (often called a *mask* or *operator*) passes over I_{in}, producing an output pixel at each discrete step as illustrated in Figure 11.2. For the linear case, "applying" the template at a given location in I_{in} means to multiply each template value by the associated underlying pixel value, and then to compute the sum of the products. This sum is the pixel value for I_{out}, and may no longer be an 8-bit quantity. It is assumed that $h = 0$ outside the specified grid. Special rules may be needed for pixels near the image borders.

Other linear filters can be implemented by changing the weights in such a template. For example, the following high-pass filters are commonly used to enhance intensity edges, which result from sharp changes in pixel values. Known as *Sobel operators*, h_1 and h_2 can be used to detect vertical and horizontal intensity gradients, respectively.

$h_1 =$

−1	0	1
−2	0	2
−1	0	1

$h_2 =$

1	2	1
0	0	0
−1	−2	−1

Larger templates are also possible, as illustrated below. Examples of images produced using these templates are shown in Figure 11.3.

$h_{LP} = 1/64 \times$

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

Low-pass filter template (see Figure 11.3b)

$h_{XY} = 1/4 \times$

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	−1	−1	0	0	0
0	0	1	0	−1	0	0	0
0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Sobel X-Y filter template (see Figure 11.3c)

After an image has been appropriately low-pass filtered, the image can be subsampled without fear of violating the Nyquist criterion. If an image is recursively filtered and subsampled, the resulting set of images can be considered a single unit and is called a *pyramid*. This data structure facilitates image analysis at different scales. Processing at the lower-resolution portion of the pyramid can be used to guide processing at higher-resolution levels. For some tasks (such as surveillance and road following) this approach can greatly reduce the overall amount of processing required.

In addition to low-pass pyramids, it is possible to generate band-pass pyramids, in which each level of the pyramid contains information from a single frequency band. A popular technique for generating these pyramids (known as Gaussian and

(a) (b) (c)

FIGURE 11.3 Example of Filtering Operations. (a) Original image. (b) Smoothed image, created by applying a low-pass filter to the original image. (c) Edge image, created by applying a Sobel XY filter. All of these images are 512 × 512 in size. The output images were obtained using 8 × 8 templates on VTSPLASH.

Laplacian pyramids) is described in [6]. A VTSPLASH implementation of a low-pass and a band-pass pyramid generator will be presented in a later section.

Neighborhood operations are not necessarily linear. For example, the output pixel value could be chosen as the *median* of the neighborhood in the input image. This nonlinear filtering operation can be expressed as follows:

$$I(r, c) = \text{median}\{ \begin{array}{lll} I(r-1, c-1), & I(r-1, c), & I(r-1, c+1), \\ I(r, c-1), & I(r, c), & I(r, c+1), \\ I(r+1, c-1), & I(r+1, c), & I(r+1, c+1)\} \end{array}$$

One advantage of this operation is reduced blurring, as compared with linear filtering. The design of a median filtering system using VTSPLASH is also described in detail in Section 11.4.

The remainder of this section presents a brief description of image processing operations that have been implemented on VTSPLASH. For example, other nonlinear operations can be implemented using the ideas of *mathematical morphology* [20, 2]. This is an algebra that uses multiplication, addition (subtraction), and maximum (minimum) operations to produce output pixels. The fundamental operations are called dilation and erosion, which cause image regions to expand and shrink, respectively. The gray-scale dilation of an image I_{in} by the structuring element h is defined as

$$I_{out} = (I_{in} \oplus h)(r, c) \equiv \max_{i,j}\{I_{in}(r-i, c-j) + h(i, j)\},$$

and erosion by h is defined as

$$I_{out} = (I_{in} \ominus h)(r, c) \equiv \min_{i,j}\{I_{in}(r+i, c+j) - h(i, j)\}.$$

These operations can be pipelined, and often serve as building blocks for higher-level processing.

Another operation that has been implemented on VTSPLASH is the 2-D discrete Fourier transform (DFT). For an $M \times N$ image, this is defined as

$$I_{out}(r, c) = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} I_{in}(k, l) \exp\left[-j2\pi \left(\frac{lr}{M} + \frac{lc}{N}\right)\right]$$

where I_{out} is composed of real and imaginary components. This can be rewritten as follows,

$$I_{out}(r, c) = \frac{1}{M} \sum_{k=0}^{M-1} \left\{ \frac{1}{N} \sum_{l=0}^{N-1} I_{in}(k, l) \exp\left[-j2\pi \left(\frac{lc}{N}\right)\right] \right\} \exp\left[-j2\pi \left(\frac{lr}{M}\right)\right],$$

which illustrates the fact that the 2-D DFT can be implemented as a sequence of 1-D DFTs. For example, the DFT of a 512 × 512 image can be obtained by first computing 512 independent 1-D DFTs (one for each row), and then computing 512 1-D DFTs of the resulting columns. This has been implemented on VTSPLASH using floating-point arithmetic [22].

The *Hough transform* [10, 13] is a technique that can be used to detect lines in an image. Assume that intensity edges have been detected, so that the Hough algorithm processes only foreground (edge) or background values. The procedure begins by initializing all values in an accumulator array to zero. For each edge point, a parametric curve is traced through the accumulator array, and each array element on the curve is incremented. Effectively, each edge point "votes" for all possible lines that pass through that point.

Referring to Figure 11.4, assume that a line is parameterized by $d = r \cos\theta + c \sin\theta$, where (r, c) represents an image location. The Hough transform is implemented as follows:

```
Algorithm Hough
Initialize all elements of accumulator array A to 0
for r = 0 to M − 1
        for c = 0 to N − 1
            if I_in(r, c) is an edge point
                for θ = 0 to 2π in steps of Δθ
                        d:= (round) (r cos θ + c sin θ)
                        A[d, θ] := A[d, θ] + 1
                end for
            end if
        end for
end for
end Hough
```

This produces the accumulator array, and has been implemented on Splash [11, 1]. The next step is to detect peaks in the array. Each local maximum represents one line in the image I_{in}. This procedure can be generalized to detect other parametric shapes, such as ellipses and polygons.

The image processing operations described above can be broadly classified into four generic classes [26]. An operation in the *combination* class takes two images

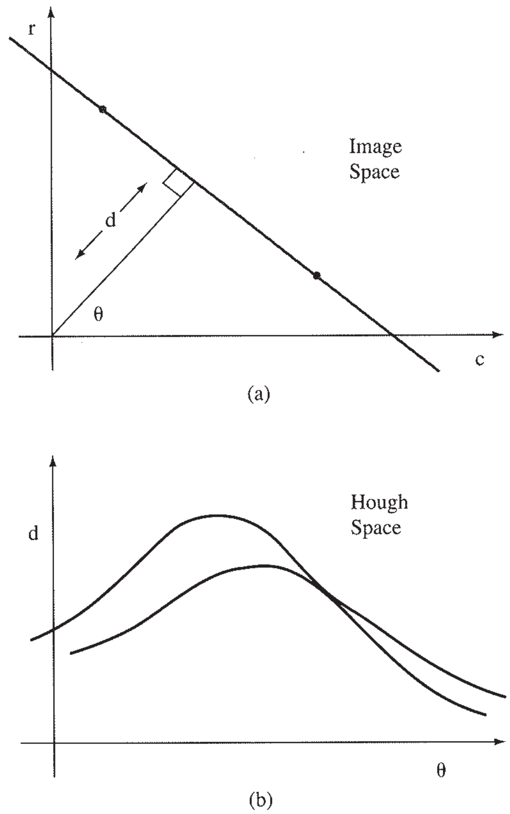


FIGURE 11.4 The Hough Transformation to Parameter Space. Edge points (r_i, c_i) in the image (a) map to sinusoids in the d-θ parameter space (b). In this example, the two sinusoids intersect at the values d and which determine the line that passes through (r_1, c_1) and (r_2, c_2).

and produces a new image of the same type. This is accomplished by combining each pair of elements from the input images into a new element. The *transformation* class accepts an image from a given class, and produces a new image in the same class. The *measurement* class reduces an image of a given type into a scalar or vector. The *conversion* class refers to those operations that take an image of a given type, and convert it into a new class.[2] Examples from each of these categories have been modeled and synthesized using the VTSPLASH system, as summarized in Table 11.1. Further descriptions of these and other image processing tasks are described in [14, 17], and [19].

These image processing tasks represent a considerable computational challenge if near-real-time operation is needed. Image pixels are typically produced and conveyed in *raster order*—pixels are presented serially, left-to-right for each image row, beginning with the top row. Consider again the 3×3 filtering operations discussed above. Although the nine neighboring pixels are spatially localized in the actual image, they are widely separated in the pixel stream from the camera. This is illustrated in Figure 11.5. For processing purposes, the straightforward approach is to store the entire input image into local memory, and then access pixels as needed

[2] Another class of operations that does not require an input image is the *generation* class, which produces a new image from scratch. This class of operations is not considered here.

TABLE 11.1 A Representative List of Image Processing Categories and Example Tasks

Class	Example image task	Description
Transformation	Convolution	Linear filtering operation.
	Median filtering	Nonlinear filter which can be used to eliminate "salt and pepper" noise.
	Morphological filtering	Nonlinear operations that alter region shapes in an image. Gray-scale *erosion* and *dilation* operations have been implemented.
Combination	Laplacian Pyramid generation	Produces an image hierarchy of decreasing image size and spatial resolution. The image for each pyramid level is formed by taking the difference of two blurred versions of the original image.
Measurement	Histogram generation	Statistical operation for computing intensity distribution of pixels in an image.
Conversion	Fast Fourier Transform	Converts an image from the spatial domain to the frequency domain.
	Hough Transform	A voting scheme that detects the presence of lines (or parametric curves) from a set of points in an image.
	Region detection and labeling	Finds connected regions in an image, and assigns a unique label to each.

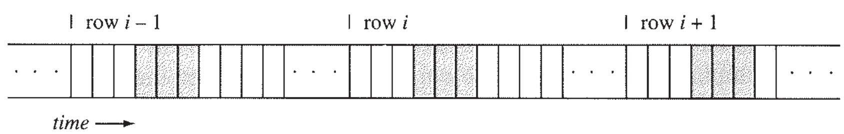


FIGURE 11.5 Example Image in Raster Order. Pixels are produced serially in row-major order. The highlighted pixels represent a single 3 × 3 image neighborhood.

to produce the output image. However, this approach introduces a latency of at least an entire image frame before the processor can begin to generate output pixels. This latency can be reduced to less than the time of n rows (for an $n \times n$ template) if the architecture is carefully designed to interleave memory reads and writes, effectively utilizing memory as a delay line. Splash 2 has been used to implement both of these processing methods. More discussion of image processing architectures can be found in [9, 16], and [24].

The default image size that is used on VTSPLASH is 512 × 512, with a pixel clock of 10 MHz. Although the rest of this chapter will discuss images in terms of monochrome light intensities, the same ideas also apply to other image types. Examples are range images, for which each pixel represents a distance value; x-ray images, where each pixel depends on object density; and computed tomography (CT) images, where each 2-D image represents a reconstructed slice of density information within a 3-D array of data.

11.4 CASE STUDY: MEDIAN FILTERING

Median filtering is a common approach for reducing noise in images [26]. Median filtering is a computational operation that replaces each picture element, or pixel, of an input image with the median value of several neighboring pixels in the image. The result is an output image that is a smoothed version of the input. Compared with traditional linear filtering, the median filter is more effective at removing impulsive noise and at smoothing an image without blurring intensity edges. Unfortunately, median filtering requires considerably more computations per pixel than linear filtering for a given neighborhood size. This is a significant problem because of the large number of pixels associated with a single image.

Rank-order filters such as the median filter are widely used for reducing noise and periodic interference patterns in images, and are useful for cleaning impulsive noise without blurring sharp edges. Implementing a median filter is computationally costly on a general-purpose platform because of the need to sort a large number of sets of pixel values repeatedly.

The median filtering operation may be stated mathematically in the following manner. Let $f_0, f_1, \ldots, f_{N-1}$ represent the intensity values for input image I_{in} within an N-point neighborhood about the point (r, c) in the image. These values are ordered so that $f_K \leq f_{K+1}$. The output image I_{out} is determined as:

$$I_{out}(r, c) = f_{(N-1)/2} \qquad \text{for odd } N$$
$$I_{out}(r, c) = \tfrac{1}{2}[f_{(N/2)-1} + f_{(N/2)}] \qquad \text{for even } N$$

In most image processing applications, rectangular neighborhoods are assumed. Conceptually, a median-filtered image is created by passing a small template over a source image. At each location of the template, the median of the image values covered by the template is selected as the corresponding value for the new image. Median filtering is therefore a neighborhood operation, characterized by repeated comparisons of neighboring pixel values.

Figure 11.6 illustrates again the concept of a 3×3 neighborhood operation. The shaded 3×3 window is assumed to "slide" over I_{in} producing an output value

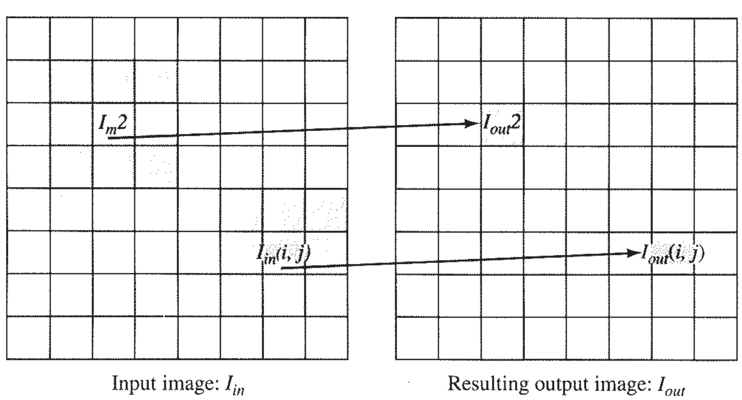


FIGURE 11.6 Concept of a 3×3 Window-Based Operation. For the median filter, the value of $I_{out}(i, j)$ is the median of the nine pixels of I_{in} which lie within the 3×3 window with center at $I_{in}(r, c)$.

for I_{out} at each location of the window. For median filtering, the value of the pixel at any location in I_{out} is the median of the nine values in the 3×3 window with center at that position in I_{in}. Two window positions are shown in the figure, with corresponding positions highlighted in I_{out}. For an input image of size 512×512, approximately 262,144 nine-point median values need to be extracted to produce I_{out}.

The median filter does a good job of estimating the true pixel values in situations where the underlying neighborhood trend is flat or monotonic and the noise distribution has flat tails. It is effective for removing impulsive noise. However, when the neighborhood contains fine detail such as thin lines, they are distorted or lost. Corners can be clipped. It can produce regions of constant or nearly constant values that are perceived as patches, streaks, or amorphous blotches. Such artifacts may suggest boundaries that really do not exist. In spite of these problems, median filtering is often an attractive alternative to traditional linear filtering. Unfortunately, the computational complexity of median filtering is much higher.

The median filter has been implemented on Splash 2 as a single-board design [23]. The design and data flow within the Splash 2 processor Array Board are shown in Figure 11.7. The design makes available all the pixels in a 3×3 window simultaneously so that a combinational sort can be performed on them. The median is then chosen from the sorted values.

Input image pixels are presented to VTSPLASH in raster order (left to right for the first image row, then repeating for each subsequent row). Pixels are presented to the first Splash 2 Processing Element at a rate of 10 MHz. The task of storing the input image is so divided that six Processing Elements are required for the purpose. Each receives the input pixel stream at the same time. This requires the input pixels to be rearranged such that every four consecutive input pixels are packed together to form a 32-bit data word. This packing of input pixels, and transferring the resulting data stream to the crossbar, is done by Processing Elements PE-1 and PE-2. The packed input data is broadcast to PE-3 through PE-8, once every four clock cycles. The effective input data rate remains unaltered.

Processing Elements PE-3 through PE-8 are responsible for storing and retrieving the image pixels in local memory. This storage is organized such that all the

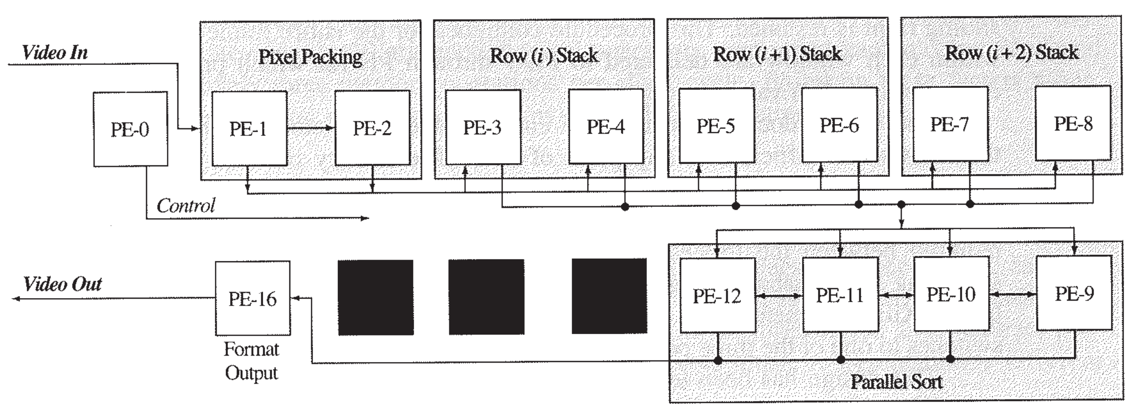


FIGURE 11.7 Communication Structure and Processing Element Layout for a Single Processor Array Board Implementation. Note that solid blocks denote unused PEs.

pixels within a 3×3 window may be accessed simultaneously. Let $I(i, j)$ represent the pixel value stored at row i and column j. Pixels are presented left to right for each row ($j = 0$ to 511), and top to bottom ($i = 0$ to 511). The first four pixels, $I(0, 0)$, $I(0, 1)$, $I(0, 2)$, $I(0, 3)$ are directed by PE-2 simultaneously to PE-3 and PE-4. $I(0, 0)$ and $I(0, 1)$ are stored in the first location of PE-3's memory while $I(0, 2)$ and $I(0, 3)$ are stored in the first location of PE-4's memory. Two pixels are packed into each 16-bit memory location. The next four pixels $I(0, 4)$–$I(0, 7)$ are stored in similar fashion in the second locations of PE-3 and PE-4.

The second row of the image is stored similarly into the local memory of PE-5 and PE-6. The third row is stored in the memory of PE-7 and PE-8. This sequence repeats, with the fourth row being stored in memories of PE-3 and PE-4, the fifth in PE-5 and PE-6, the sixth in PE-7 and PE-8, and so on, until the entire image has been captured.

The retrieval of the stored pixels begins as soon as three rows have been received. As soon as the first three rows are stored in the memory of PE-3 through PE-8, all six PEs (PE-3–PE-8) perform a read operation from the first location of their local memory. With two pixels packed within each memory location, the six PEs are capable of concurrently accessing a total of 12 pixels. At this point, data corresponding to a 3×4 window is available for processing. The 3×4 window referred to here lies within the range $i = 0$ to 2 and $j = 0$ to 3. Two complete 3×3 windows lie within this 3×4 window and may therefore be processed at once.

The two rightmost columns of data in the window ($j = 2$ and 3) are stored in registers internal to the FPGAs. This storage helps create two additional 3×3 windows every time a 3×4 window is formed.

In the subsequent read cycle, four new pixels for each of the first three rows ($j = 4$ to 7) are read from memory. Since two columns have been stored in internal FPGA registers, the effective window size is 3×6 instead of 3×4. Four 3×3 windows may be formed from this window and thus four median values may be computed simultaneously.

This process continues with the 3×4 window sliding four pixels to the right in every read operation. Once the window reaches the extreme-right border of the image ($j = 488$ to 511), it "wraps" around in a "snake-like" fashion such that it moves one row to the bottom and starts from the leftmost border. The process of sliding right is resumed. This procedure continues for the entire frame and the pixels within each window are delivered to PE-9 through PE-12, which process them to compute a median value.

The design does not require the entire image to be stored in memory. Only three rows are sufficient at any point of time. The latency between the input and output frames is approximately three rows—a latency that is typically achieved by dedicated image processing hardware. A substantial number of data transfers are required between the Processing Elements on the Array Board, and this requires switching the crossbar configuration every clock cycle. This switching is controlled by the Xilinx element PE-0. PE-0 is programmed such that in every clock cycle, it switches to one of the three possible crossbar configurations, which are user-specified.

This design has been tested using the image shown in Figure 11.8. Noise was artificially introduced into the input image, and has been removed in the filtered image produced by Splash 2. Also, careful observation reveals contours or regions of small plateaus formed in the resulting image. This is another result that is expected

(a) (b)

FIGURE 11.8 (a) Input test image for median filtering. This is a 512 × 512 gray-scale image that is presented to Splash 2. To demonstrate the noise-cleaning effect of median filtering, noise is deliberately introduced in the image. This is seen as black and white spots. (b) Median-filtered image obtained from Splash 2. The noise that was introduced in the original image has been filtered out. This demonstrates the noise-cleaning property of the median filter.

by median filtering. The image obtained by simulation using a C program compares well with the result image obtained from Splash 2, differing only in the pixel values at the frame edges. This difference arises because the border effect is ignored in the Splash 2 design.

With a 10 MHz clock on VTSPLASH (the video pixel rate), the time to process one frame is 0.027 seconds. The same task, written in C, and compiled with the appropriate optimizations, requires 8.0 seconds on a SPARCstation-2 and 3.75 seconds on a SPARCstation-10. The implementation presented here performs a number of arithmetic and memory operations in parallel. Although this is difficult to quantify, there are roughly 39 arithmetic/logical operations performed each clock cycle,[3] and effectively three memory operations per clock cycle. Based on these factors alone, this application effectively performs 420 million operations per second.

11.5 CASE STUDY: IMAGE PYRAMID GENERATION

Multiresolution and multirate image processing techniques have become increasingly popular over the past decade because of the advantages of processing image data at different scales. A basic data structure used in multiresolution and multirate processing is the image pyramid, which is a complete image representation at different

[3]In a hardware implementation, the process of identifying "operations" that correspond to instructions found in typical microprocessors is somewhat subjective. In this approximation, only major "word"-wise operations (such as *add* or *shift*) were considered.

levels of resolution. An image pyramid is constructed by recursively applying two basic operations—filtering and subsampling—to an image, creating a set of images of decreasing size and spatial resolution. Filtering is performed to convolve the input image with a family of local, symmetric smoothing functions. Subsampling then produces samples for the images at the next-higher scale. The two most common image pyramids are the Gaussian (low-pass) and the Laplacian (band-pass) pyramids [6].

11.5.1 Gaussian Pyramid

The sequence of images $g_0, g_1, \ldots, g_{k-1}$ as shown in Figure 11.9a is called a Gaussian pyramid. A weighting function that resembles the Gaussian probability distribution is applied to each pixel neighborhood of the original video image g_0 to generate the lower-resolution image g_1, which is used in turn to generate g_2, and so on. The level-to-level filtering and resampling can be expressed as a function REDUCE as shown below:

$$g_k = \text{REDUCE}(g_{k-1}) \qquad (11.1)$$

where each pixel value in g_k is obtained by a weighted sum of pixels from g_{k-1}, computed over a 5×5 neighborhood as follows [18]:

$$g_k(i, j) = \sum_{m=-2}^{2} \sum_{n=-2}^{2} \omega(m, n) g_{k-1}(2i - m, 2j - n) \qquad (11.2)$$

To simplify the computational requirements, the 5×5 weighting function ω is often chosen to be separable into two one-dimensional filters: $\omega(m, n) = \omega_x(m)\omega(n)$.

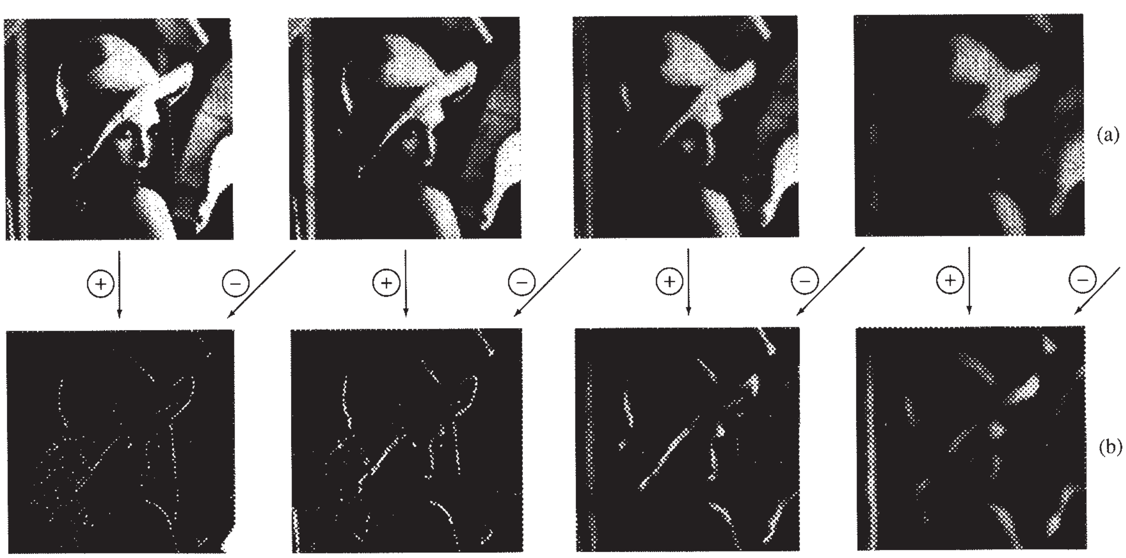


FIGURE 11.9 Example Data Produced from (a) a Gaussian Pyramid, and (b) a Laplacian Pyramid (from [27]).

The function REDUCE in Equation (11.2) is then split into two functions, REDUCEX and REDUCEY:

$$g_{k,x}(i,j) = \text{REDUCEX}(g_{k-1}) = \sum_{m=-2}^{2} \omega_x(m) g_{k-1}(2i-m, j)$$
$$g_k(i,j) = \text{REDUCEY}(g_{k,x}) = \sum_{m=-2}^{2} \omega_x(n) g_{k,x}(i, 2j-n) \quad (11.3)$$

The 1-D weighting function in the vertical direction, ω_y, is usually the transpose of the function in the horizontal direction, ω_x. The functions ω_x and ω_y are constructed so that it is normalized ($\sum_{i=-2}^{2} \omega(i) = 1$), symmetric ($\omega(i) = \omega(-i)$), and the equal contribution rule [25] which requires that $a + 2c = 2b$, where $a = \omega(0)$, $b = \omega(-1) = \omega(1)$, and $c = \omega(-2) = \omega(2)$. Although other solutions are possible, these three constraints are satisfied when $\omega(0) = a$, $\omega(1) = 1/4$, and $\omega(2) = 1/4 - a/2$. The equivalent weighting function is particularly Gaussian-like when a is around 0.4. For implementation in digital logic, it is convenient to choose $a = 3/8$, $b = 1/16$, and $c = 1/4$.

Since the denominators of all weighting factors are powers of two, the multiplication of image pixels by the weighting factors can be simply implemented using binary shift operations. For instance, a pixel multiplied by 3/8 is the sum of the value shifted two places to the right plus the original value, all shifted three places to the right.

To maintain numerical accuracy, the summation elements have been expanded to 12 bits each. Four bits with values of 0 are appended to the right of each image pixel value before computation. The eight most significant bits of the final result are maintained.

11.5.2 Two Implementations for Gaussian Pyramid on Splash 2

Figure 11.10a shows the block diagram of a five-chip pyramid generation architecture that has been developed for Splash 2 [1, 7]. This implementation is based on the recirculating pipeline structure, and is designed to produce five levels of pyramids (g_0 through g_5). Although compact, this architecture is capable of converting only every other image frame into pyramid form (15 frames per second). The Control Element PE-0 buffers image pixels, and passes the data to Processing Element PE-1 through the crossbar. The processing steps of this architecture are horizontal convolution by ω_x (Processing Element PE-1), vertical convolution by ω_y (Processing Elements PE-2 and PE-3), and recirculating and output image production (Processing Element PE-4).

The Control Element PE-0 broadcasts image pixels, representing g_0, through the crossbar to Processing Elements PE-1 through PE-3, which compute the first level of the Gaussian pyramid, g_1. Image data is recirculated through the crossbar to PE-1, and processed through the same path to form the higher pyramid levels. Two different crossbar configurations are used to multiplex the original image data and feedback pyramid data. PE-0 controls the crossbar configuration, which is used during processing.

Device PE-1 receives image data from either PE-0 or PE-4 through the crossbar, computes the convolution by ω_x, and passes the result to PE-2. Resampling in

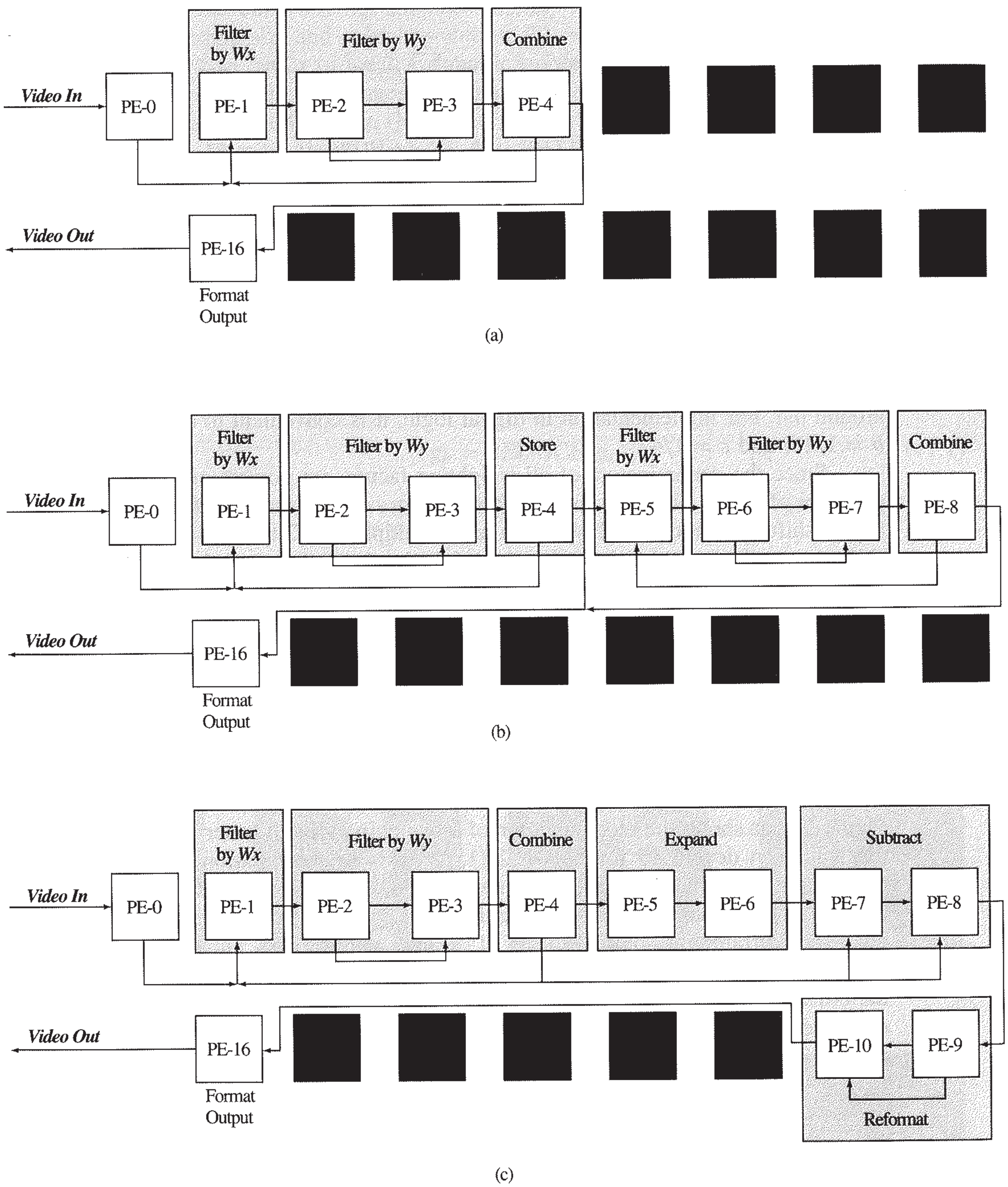


FIGURE 11.10 Examples of the Communications Structure and Partitioning of One-Board Pyramid Applications. a) simple five-level Gaussian Pyramid generator, b) Gaussian Pyramid generator using the hybrid pipeline architecture, and c) five-level Laplacian Pyramid generator.

the horizontal dimension is performed during the convolution to eliminate half of the computations. The image data that is passed to PE-2 has half of the pixels per image row.

The image data is presented into Splash 2 one row at a time in raster order. The 8-bit image pixels that are presented to PE-1 are grouped so that four pixels are passed simultaneously on the crossbar. Four control bits on this data path are appended to indicate data validity and the pyramid level.

PE-2 and PE-3 together implement the convolution by ω_y. Unlike the convolution in the horizontal direction, the five pixels required by each computation are not presented in the same image row, but in five consecutive rows. The image data, therefore, needs to be stored in a delay line, which is implemented using the external RAMs. One memory write and four memory reads are needed for sequencing the data for each 5×1 convolution. Only one memory write and two memory reads are allowed in four Splash 2 cycles because of access constraints. PE-2 computes three of the five partial sums, and passes the 12-bit partial result directly to PE-3. PE-3 performs the remaining three partial sums, and passes the rounded 8-bit value to PE-4.

PE-4 resamples the image data in the vertical dimension to reduce the number of pixels per image-column by half. The data are then recirculated to PE-1 through the crossbar to form the next level of the pyramid. Each pyramid level is also made available to the next Processing Element, PE-5, for further analysis.

11.5.3 The Hybrid Pipeline Gaussian Pyramid Structure

The block diagram of a nine-chip hybrid structure of a Gaussian pyramid generator is shown in Figure 11.10b. The original image pixel (g_0) are passed to PE-1 directly from the input stream, and are processed through Processing Elements PE-1 through PE-4 to form the first-level Gaussian pyramid, g_1. Processing Elements PE-5 through PE-8 generate the remaining four levels of the pyramid. PE-9 takes data from PE-4 and PE-8 to form the resulting pyramids.

The hybrid implementation requires five more PEs than the recalculating implementation. The two stages comprised of PE-1 through PE-4 and PE-5 through PE-8 are very similar in structure. The key advantage of this algorithm (at the cost of four additional PEs) is that it is capable of generating Gaussian pyramids in real time (30 frames per second).

11.5.4 The Laplacian Pyramid

The Laplacian pyramid as illustrated in Figure 11.9b is a sequence of difference images, in which each image is the difference between two successive Gaussian levels. Two types of Laplacian pyramids are in common use: the filter-subtract-decimate (FSD) structure and the reduce-expand (RE) structure [6].

The FSD Laplacian is formed by subtracting a filtered image of the next-higher Gaussian pyramid level from the same level of the pyramid image. The kth level of the FSD Laplacian pyramid can be expressed as,

$$L_k^{FSD}(i, j) = g_k(i, j) - g_{k+1}^F(i, j) \tag{11.4}$$

where g_{k+1}^F is the $(k+1)$th level of the filtered Gaussian image before subsampling.

The RE pyramid generation structure includes two basic operations: image expansion and image subtraction. The EXPAND operation can be regarded as the reverse of the REDUCE function in Gaussian pyramid generation. First, the image size is doubled by inserting a pixel with a gray level of '0' between two successive pixels in every row and column. The expanded image is then convolved by the same Gaussian-like weighting function. As was done for the REDUCE function, the EXPAND operation is split into two 1-D identical convolutions applied to the image in both horizontal and vertical direction. The 1-D operation can be expressed as below:

$$g^i(x) = 2 \sum_{m=-2}^{2} \omega(m) g^e(x-m) \tag{11.5}$$

and

$$g^e(x) = \begin{cases} g\left(\frac{x}{2}\right) & \text{if } x \text{ is even} \\ 0 & \text{if } x \text{ is odd} \end{cases} \tag{11.6}$$

where $g(x)$ is the Gaussian pyramid image, and $g^i(x)$ and $g^e(x)$ are the 1-D interpolated and expanded image, respectively. The above equations can also be represented in a more explicit way:

$$g^i(x) = \begin{cases} 2 \times \left[\omega(-2) \times g\left(\frac{x}{2}+1\right) + \omega(0) \times g\left(\frac{x}{2}\right) + \omega(2) \times g\left(\frac{x}{2}-1\right)\right], & \text{if } x \text{ is even} \\ 2 \times \left[\omega(-1) \times g\left(\frac{x+1}{2}\right) + \omega(1) \times g\left(\frac{x-1}{2}\right)\right], & \text{if } x \text{ is odd} \end{cases} \tag{11.7}$$

Replacing the weighting factors $(\omega(-2), \ldots, \omega(2))$ with their values $\left[\frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16}\right]$, the equation can be simplified as follows:

$$g^i(x) = \begin{cases} \frac{1}{8} \times \left[g\left(\frac{x}{2}+1\right) + g\left(\frac{x}{2}-1\right)\right] + \frac{3}{4} \times g\left(\frac{x}{2}\right), & \text{if } x \text{ is even} \\ \frac{1}{2} \times \left[g\left(\frac{x+1}{2}\right) + g\left(\frac{x-1}{2}\right)\right], & \text{if } x \text{ is odd} \end{cases} \tag{11.8}$$

The odd-numbered pixel of the expanded image is equal to the weighted sum of two pixels in the Gaussian pyramid, and the even-numbered pixel is the weighted sum of three pixels, for instance pixels 1 and 4. The 1-D EXPAND operation can be considered as functions of 2-by-1 convolutions and 3-by-1 convolutions, with weighting functions of $\left[\frac{1}{2}, \frac{1}{2}\right]$ and $\left[\frac{1}{8}, \frac{3}{4}, \frac{1}{8}\right]$, respectively. Both weighting functions are normalized and symmetric as well. The edge pixels 0, 8, and 9 are not defined in Equation (11.4). In this design, the first and last calculated values, pixels 1 and 7, are duplicated to form the edge.

Once the pyramid is expanded to have the same size as the next-higher resolution pyramid, the *subtraction* operation is applied to obtain one Laplacian pyramid level. The function is expressed as:

$$L_k^{RE}(i,j) = g_k(i,j) - g_{k+1}^{int}(i,j) \tag{11.9}$$

where g^{int} is the interpolated image constructed from g^e.

11.5.5 Implementation of the Laplacian Pyramid on Splash 2

The Laplacian pyramid-generation system consists of two major parts: Gaussian pyramid generation, and image subtraction. The system uses the recirculating pipeline structure, as presented in the previous section, to generate a Gaussian image pyramid. After the Gaussian pyramid is generated from Processing Elements PE-0 through PE-4, the Laplacian pyramid is computed by Processing Elements PE-5 through PE-10, as shown in Figure 5.2. The data is passed directly to Processing Element PE-5, and to PE-7 and PE-8 through the crossbar. Devices PE-5 and PE-6 implement the EXPAND operation in the horizontal and vertical directions, respectively. The pixel-by-pixel SUBTRACTION operation is then implemented in chips PE-7 and PE-8 to generate a difference image. PE-9 and PE-10 reformat the images for output.

As described in the previous section, the data output from PE-4 to PE-5 is the image data directly from the "XP_Right" port of device PE-3. The 36-bit-wide bus carries only 20 bits of useful information: two 8-bit image pixels and four control bits. Since PE-3 does not perform the subsampling function in the vertical direction, the even-numbered rows of the image data are ignored in future data processing. A depiction of this implementation is given in Figure 11.10c.

11.6 PERFORMANCE

This section provides a quantitative summary of the performance of VTSPLASH for the operations discussed in the previous section. The computational properties, communications architectures, and required resources vary significantly from one application to the next. All of these examples operate at the pixel clock rate of 10 MHz with 512×512 images. Many of the applications presented here have been implemented using a pipeline architecture. The pipeline accepts digitized image data in raster order, often directly from a camera, and, in most cases, produces output data at the same rate, possibly with some latency. Many of these applications can be chained together to form higher-level image processing functions.

Simplified block diagrams illustrating the partitioning and communication architecture for selected tasks are shown in Figure 11.11. For example, Figure 11.11a shows the architecture for a region detection and labeling application [18]. This application analyzes an image to distinguish foreground objects from background through thresholding, and then for each foreground image, a unique label is assigned. This task is a useful front end for applications such as recognition, industrial inspection, and tracking. After the image is appropriately thresholded, an initial estimate is made of the disjoint regions in the image by the block labeled *Pass 1 Labeling*. It may be subsequently discovered that regions that were initially disjoint are actually contiguous. Such regions need to be merged and assigned the same label. This is accomplished in the following two blocks, *Pass 2 Merging (EVEN)* and *Pass 2 Merging (ODD)*.

Conventional performance-benchmarking techniques are at best awkward when applied to custom computing machinery. Figure 11.12 illustrates graphically the computational performance of each of these tasks executing on the VTSPLASH platform. In the figure, the application name is listed to the left of the graph. The

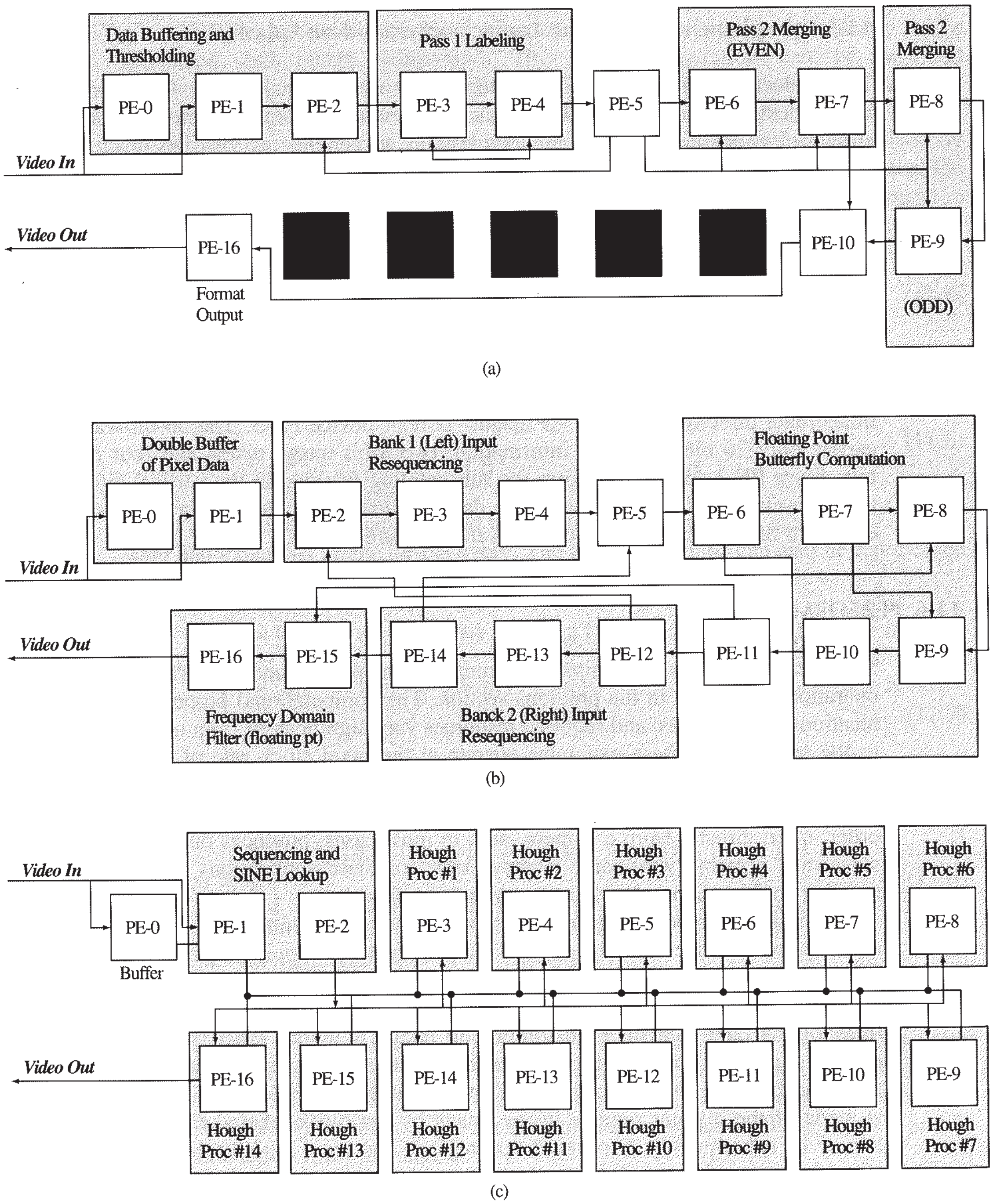


FIGURE 11.11 Examples of the Communications Structure and Partitioning for Examples that Use Only One Splash 2 Processor Array. a) region detection and labeling, b) FFT (forward transform), and c) Hough transform. Solid squares at Processing Element sites denote unused Processing Elements.

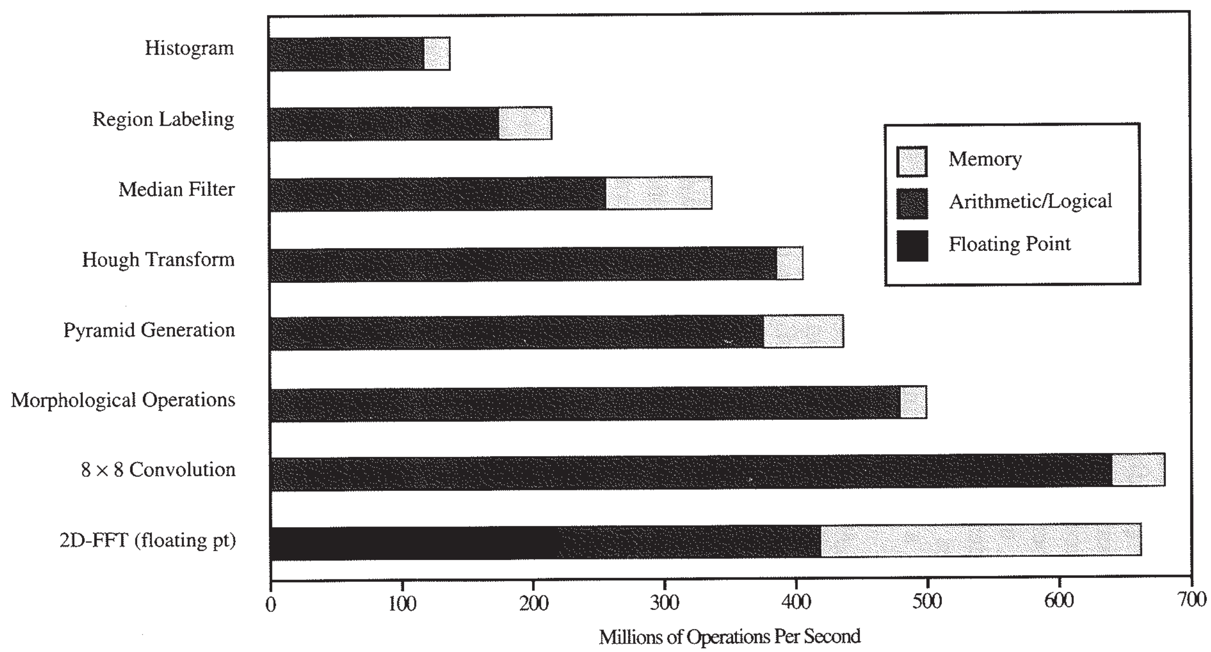


FIGURE 11.12 Approximate Performance of Image Processing Tasks

performance bar associated with each task consists of two or three components. The first component *(arithmetic/logical)* is an appraisal of the number of general-purpose operations performed, on average, per second. (These are operations that are likely to be found in the repertory of common RISC processors, such as MULTIPLY, XOR, or COMPARE.) This number, when divided by the pixel clock frequency of 10 MHz, gives an indication of the average number of the easily discernible arithmetic and logic function units (word parallel) that are active in each task. The second component of the performance bar provides an estimate of the number of storage references (memory accesses) performed by the task per second. The third component represents the number of floating-point operations. All of the tasks, except for the 2D-FFT application, use fixed-point operators. The pixel calculations for the 2D-FFT task utilize custom-designed floating-point arithmetic. The combination of these three components provides a basis for quantifying the computational load of each of the tasks, and provides a rough estimate of the number of operations performed each second.

The operating speed for an application is under the control of the designer, and depends upon critical path delays in the implementation. The Splash 2 processor features a programmable system clock that can be varied under software control from zero to 40 MHz. The tasks developed in this project were made to satisfy the *minimum* criteria of operating at the pixel data rate of 10 MHz. Because of limitations of the image data source, the listed applications were tested only at this rate. It is feasible that some of these tasks operate well beyond this clock frequency.

In addition to quantifying the number of operations per second, it is useful to consider how fast computations are performed relative to the input image frame rate of 30 Hz. Some of the tasks are completed during one frame time (histogramming, median filtering, Gaussian pyramid generation, and gray-scale morphological operations). Others require two image frame times (region labeling, 8×8 convolution, and Laplacian pyramid generation). The FFT implementation can completely process two 512×512 images per second (or 128×128 images at 30 frames per second) [21]. The time to complete the Hough transform is image-dependent; the implementation shown in Figure 11.11c distributes equal portions of an input image to separate PEs that process in parallel.

Another method of benchmarking the performance is to compare with contemporary machines. Comparisons were made with a general-purpose workstation (a Sun SPARCstation-10). The VTSPLASH applications run between 10 to 100 times faster than the same application written in C and executed on the SPARC workstation. A number of commercial machines exist that have been designed specifically for image processing. The Datacube MaxVideo 200 [8], for example, consists of several functional units that have been carefully tuned to perform common image-processing tasks. In most cases, for the specific tasks that are implemented by the application-specific hardware, the VTSPLASH system is outperformed. For example, the MaxVideo 200 can perform 8×8 convolution four times faster than the existing VTSPLASH implementation. The motivation of the custom-computing approach, therefore, is *not* to provide the fastest possible performance for a given task. As illustrated by VTSPLASH, the strength of this approach is the ability of the system to be reconfigured to provide high performance for a wide range of tasks. The performance of application-specific systems diminishes quickly for tasks that are not directly supported in hardware.

11.7 SUMMARY

Reconfigurable computing platforms, such as Splash 2, can readily adapt to meet the communication and computational requirements of a wide variety of applications. With the addition of input/output hardware, we have demonstrated that general-purpose custom computing machines are well suited for many meaningful image processing tasks. Such platforms are excellent testbeds for prototyping high-performance algorithms. The custom computing platform can be viewed not only as a general-purpose computing engine, but also as:

- a medium for hardware/software codesign
- a VHDL accelerator
- a testbed for rapid prototyping

Furthermore, the platform is multi-use since it can be reconfigured from one task to another by downloading a hardware-personalization database.

Applications operational on the VTSPLASH laboratory system include:

- 2-D Fast Fourier Transform (using floating point)
- Expandable 8 × 8 convolver (with on-line filter adjustment)
- Pan and zoom
- Median filtering
- Morphologic operators
- Histogram and graphical display
- Region detection and labeling

Splash is representative of the state of the art in custom computing processors—both in hardware capabilities and software support—yet it requires a substantial time investment to develop an application. To make this class of machinery more widely accepted and cost-effective, methods must be developed to reduce application development time. There are several promising endeavors that focus on this issue [3, 5, 12, 15].

ACKNOWLEDGMENTS

This project has been a success only because of the hard work of the entire VTSPLASH team at Virginia Tech. The team has included several graduate students, some of whom have graduated and taken leave for broader horizons. The major players have been Luna Chen, Robert Elliott, Brad Fross, Jeff Nevits, James Peterson, Ramana Rachakonda, Nabeel Shirazi, Adit Tarmaster, and Al Walters. The authors also gratefully acknowledge support and guidance from Jeffrey Arnold and Duncan Buell from the Supercomputing Research Center, and John McHenry.

REFERENCES

[1] L. Abbott et al., "Finding Lines and Building Pyramids with Splash-2," *Proc. IEEE Workshop FPGAs for Custom Computing*, CS Press, Los Alamitos, Calif., 1994, pp. 155–163.

[2] A.L. Abbott, R.M. Haralick, and X. Zhuang, "Pipeline Architectures for Morphologic Image Analysis," *Machine Vision and Applications*, Vol. 1, No. 1, 1988, pp. 23–40.

[3] L. Agarwal, M. Wazlowski, and S. Ghosh, "An Asynchronous Approach to Efficient Execution of Programs on Adaptive Architectures Utilizing FPGAs," *Proc. IEEE Workshop FPGAs for Custom Computing*, CS Press, Los Alamitos, Calif., 1994, pp. 111–119.

[4] P.M. Athanas and A.L. Abbott, "Processing Images in Real Time on a Custom Computing Platform," in R.W. Hartenstein and M.Z. Servít, eds., *Field-Programmable Logic: Architectures, Synthesis, and Applications*, Springer-Verlag, Berlin, 1994, pp. 156–167.

[5] P. Athanas and H. Silverman, "Processor Reconfiguration through Instruction-Set Metamorphosis: Architecture and Compiler," *Computer*, Vol. 26, No. 3, Mar. 1993, pp. 11–18.

[6] P.J. Burt and E.H. Adelson, "The Laplacian Pyramid as a Compact Image Code," *IEEE Trans. Comm.*, Vol. COM-31, No. 4, Apr. 1983, pp. 532–540.

[7] L. Chen, "Fast Generation of Gaussian and Laplacian Image Pyramids Using an FPGA-based Custom Computing Platform," master's thesis, Virginia Polytechnic Inst., Blacksburg, Va., 1994.

[8] Datacube, Inc., The MaxVideo 200 Reference Manual, Datacube, Inc., Danvers, Mass., 1994.

[9] P.M. Dew, R.A. Earnshaw, and T.R. Heywood, eds., *Parallel Processing for Computer Vision and Display*, Addison-Wesley, Reading, Mass., 1989.

[10] R.O. Duda and P.E. Hart, "Use of the Hough Transform to Detect Lines and Curves in Pictures," *Comm. of the ACM*, Vol. 15, 1972, pp. 11–15.

[11] R. Elliott, "Hardware Implementation of a Straight Line Detector for Image Processing," master's thesis, Virginia Polytechnic Inst., Blacksburg, Va., 1993.

[12] M. Gokhale and R. Minnich, "FPGA Computing in a Data Parallel C," *Proc. IEEE Workshop FPGAs for Custom Computing*, CS Press, Los Alamitos, Calif., 1993, pp. 94–101.

[13] P.V.C. Hough, "A Method and Means for Recognizing Complex Patterns," U.S. Patent No. 3,069,654, 1962.

[14] B. Jahne, *Digital Image Processing*, Springer-Verlag, New York, 1991.

[15] Q. Motiwala, "Optimizations for Acyclic Dataflow Graphs for Hardware-Software Codesign," master's thesis, Virginia Polytechnic Inst., Blacksburg, Va., 1994.

[16] R.J. Offen, *VLSI Image Processing*, McGraw-Hill, New York, 1985.

[17] W.K. Pratt, *Digital Image Processing*, Wiley, New York, 1978.

[18] R. Rachakonda, "Region Detection and Labeling in Real-time Using a Custom Computing Platform," master's thesis, Virginia Polytechnic Inst., Blacksburg, Va., 1994.

[19] A. Rosenfeld and A. Kak, *Digital Picture Processing*, 2nd ed., Academic Press, New York, 1982.

[20] J. Serra, *Image Analysis and Mathematical Morphology*, Academic Press, London, 1982.

[21] N. Shirazi, "Implementation of a 2-D Fast Fourier Transform on an FPGA-based Computing Platform," master's thesis, Virginia Polytechnic Inst., 1995.

[22] N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating-Point Arithmetic on FPGA-based Custom Computing Machines," *Proc. IEEE Symp. FPGAs for Custom Computing*, CS Press, Los Alamitos, Calif., 1995, pp. 155–162.

[23] A. Tarmaster, "Median and Morphological Filtering of Images in Real Time Using an FPGA-based Custom Computing Platform," master's thesis, Virginia Polytechnic Inst., Blacksburg, Va., 1994.

[24] L. Uhr, ed., *Parallel Computer Vision*, Academic Press, New York, 1987.

[25] G. VanDerWal and P. Burt, "A VLSI Pyramid Chip for Multiresolution Image Analysis," *Int'l J. of Computer Vision*, Vol. 8, No. 3, 1992, pp. 177–189.

[26] R. Vogt, *Automatic Generation of Morphological Set Recognition Algorithms*, Springer-Verlag, New York, 1989.

[27] Xilinx, Inc., *The Programmable Gate Array Data Book*, Xilinx, Inc., San Jose, Calif., 1994.

CHAPTER 12

The Promise and the Problems

Duncan A. Buell and Jeffrey M. Arnold

The time has come to reflect upon what we have done. The soldering irons have grown cold on the workbenches, the celebration cake has long been eaten, and even the T-shirts are fading from too many launderings. What have we learned? Where did we go right? Where did we go wrong? Have suppositions been confirmed as facts or debunked as myths? Most important, for it is the whole basis for research, what from our experience might prove valuable to the next builders of such hardware?

12.1 SOME BOTTOM-LINE CONCLUSIONS

12.1.1 High Bandwidth I/O Is a Must

This will come as no surprise to anyone in the traditional high-performance computing business, but in our situation, the rationale is slightly different. We have, in a CCM, relatively little state that can be retained in the processor portion of the machine. To achieve high performance, then, one must have an application that requires extensive computation localized on a very small amount of data or a computation that requires relatively little state but is "compute-intensive" because it must be done to a relatively large volume of data. The RSA encryption/decryption algorithm done by Shand et al. [3] at the DEC Paris lab—modular exponentiation of 512-bit integers with 512-bit exponents—is an example of the former kind of application but we have found such applications, in general, to be rare. The latter category of applications, including signal processing, image processing, data compression, and the like, appear to predominate. To handle such applications, it must be possible to get data to the CCM at a rate that permits the FPGAs to demonstrate their computational superiority.

Another issue that contributes to the desire to operate on large sets or continuous streams of data is the relatively high cost of loading an application "program" onto the FPGA. A Xilinx XC4010 takes about 22 msec to configure (180, 000 bits at 8 MHz). With system overhead from a workstation disk, this can approach 100 msec. At a clock speed of 20 MHz, 100 msec is 2 million cycles lost to reconfiguration. If each configuration of the FPGA ran for as many as 2 million cycles, the CCM would be utilized only half the time; to achieve 90 percent utilization, each configuration would need to execute on the order of 18 million cycles.

A corollary of the conclusion that I/O bandwidth is important is that I/O from the CCM to the outside world, and not just to the host computer, is essential. The 4 Mbytes per second or so that can be delivered from a SCSI disk is not enough. In the world of supercomputers, it is often observed that one of the few attributes distinguishing a supercomputer from a high-performance workstation is the speed at which data can be delivered from disk to processor. CCMs are unlikely to be designed to connect to supercomputers, if only because the small number of supercomputers makes it difficult for a commercial CCM industry to develop around them. We believe, therefore, that for CCMs to become commercially successful there must be a model of data flow and control similar to that of Splash 2: in addition to the usual programming and control lines to the host (workstation), there must be an ability to take data from some other source at rates much higher than workstation disks allow. We remark that our conclusion here seems to be consistent with the thoughts behind and design of the DEC systems.

12.1.2 Memory Is a Must

We have reasoned that a CCM like Splash 2 needs high I/O bandwidth because the computations must be relatively simple and must require relatively little state. Therefore, in order to be useful, the CCM must process a large volume of data. Our conclusion that it is important to have as much memory as possible as close to the FPGAs as possible stems from a similar line of reasoning. The Processing Elements one designs into the FPGAs must be relatively simple; the FPGAs are not yet large enough to accommodate complex objects, and they operate at speeds that are slow by microprocessor standards, so multiple-tick state machines are not going to provide a performance advantage unless significant pipelining is possible. It has been our experience that including memory for lookup tables and similar augmentations of processor state is absolutely vital to obtaining high performance. Memory is essential, and the more memory the better, because it permits, among other things, a fast horizontal encoding that requires little logic to implement, instead of a vertical encoding that takes either more complex logic or more pipeline steps.

We point out here, as was mentioned once before, that some of the lookup tables one might want to use would be much larger than could reasonably be implemented in any system. A lookup table for an 8-bit by 8-bit multiplier requires only $\frac{1}{8}$ Mbyte of memory, for example, but an only slightly less modest 12-bit by 12-bit multiplier requires 48 Mbytes. We further mention that the memory *structure* can also be important. We were pin-limited in Splash 2 and coupled one memory to one FPGA. As FPGAs accommodate larger and larger designs on a single chip, the probability will grow that more than one part of a given chip's design will need to access memory in the same clock period. The data stream-oriented computations on Splash 2 tended to have many small computational units in a pipeline. It can easily happen that each of these needs its own lookup table but more than one exists on a single FPGA, making a single memory port the bottleneck.

To some extent our conclusion here differs from what one might deduce from the work at DEC, but we remain skeptical of designs in which the FPGAs and the memories lie in separate clusters. There has been work and there seems to be continuing interest in single chips or in multi-chip modules that closely couple programmable logic and memory. Either arrangement would enlarge the processor state without continuing the current limitations, faced by Splash 2 and all other present systems, of insufficient pins for the memory bandwidth desired plus the inherent loss of speed in having to go off-chip for memory references. The disadvantage of this approach (at least the single-chip approach) is that the amount of memory that can be integrated with the processor is severely limited. This implies the need for a hierarchical memory, that is, a larger external backing store in addition to the on-chip memory, which would now function much as a cache functions in traditional processor architectures.

12.1.3 Programming Is Possible, and Becoming More So

We began Splash 2 with the firm belief that it would be possible to program Splash 2 from a high-level language, but without any clear notion of exactly *how* this would be accomplished. Our belief has not turned out to be a delusion, and the clear ideas of how to accomplish the desired ends came to us as we progressed in the project. There were questions about whether an appropriate subset of VHDL could be identified as the high-level "programming language." There were questions about whether the VHDL environment provided by vendors would provide the support we needed and, if not, whether our own augmentations could be made. There were a number of questions about the ability to sequence the vendors' tools into a compilation process. In part due to our sponsorship of work on the Synopsys FPGA Compiler and in part as a consequence of more general interest in CCMs, the path from high-level VHDL to Xilinx bitstream files is much smoother than it was three years ago. Xilinx, on the one hand, has raised the level at which their software supports design—the XBLOX tool allows circuit designers to use much larger building blocks of registers, sequencers, and the like, instead of constructing them individually from CLBs. From the top down, Synopsys has made a serious commitment to target the architecture of FPGAs in the technology-mapping phase of logic synthesis so that the resources of Xilinx (and other) FPGAs can be used efficiently and achieve performance closer to that attainable with handcrafted designs. There is now a reasonably smooth process from VHDL to Xilinx chips that yields acceptably high performance, and the situation will no doubt continue to improve in the future.

12.1.4 The Programming Environment Is Crucial

We have asserted that programming of CCMs is in fact possible. We now maintain further that the great effort we expended to create a complete programming environment has been crucial to users' acceptance of the fact that a CCM is to be viewed as a "computer."

Users of modern computer systems expect an interactive programming environment. They expect to be able to compile programs quickly, test them on sample data, step them through a debugger, and examine the resulting output. With many experimental hardware systems, performing these tasks on the hardware itself can be quite difficult. Complicating the usual problems of dealing with experimental hardware (which one might imagine to be of questionable reliability) is the very significant problem for Splash 2 and for similar CCMs of the time required for logic synthesis and the placement and routing of the netlist onto the Xilinx chips. In the absence of the simulation environment that allowed programs to be written and debugged until they were functionally correct, we doubt that many of our applications would have been completed. Certainly we feel that none of the "users" (as distinct from the "true believers") would have been willing to follow through to a completed application without the full panoply of simulation and development tools available to them.

A further reason to stress this point is that although, on the one hand a solid programming environment is an obvious desideratum, its achievement requires the cooperation of vendors. In order for T2 to be successful in a debugging mode, it was necessary that T2 be able to associate with the objects of the synthesis process the VHDL objects named within the program; otherwise, it would not be possible for T2 to examine the state in the FPGAs for debugging purposes. Similarly, although users need not ordinarily be concerned with information at the bitstream file level, those who would write system software and programming development tools may have occasion to need some of this information, at least the placement or mapping of flip-flops to CLBs and the ability to extract the flip-flop state from the chip in readback mode for debugging. Certainly, if one is to envision a CCM acting as a closely connected coprocessor instead of as an attached processor "at cable's length" like Splash 2, some details are also necessary. One concept being explored is the idea of swapping parts of a design on an FPGA in and out, in the way that code is swapped in and out of virtual memory. This will require that the systems software writer have access to information about the location of the portions of the design to be swapped out, and the I/O paths in and out of those regions of the FPGA. Swapping hardware also implies the need to constrain the physical mapping phase of compilation to lay the logic out in particular shapes, or use only particular portions of the chip.

12.2 TO WHERE FROM HERE?

Throughout the Splash 2 project, we were asked the obvious question, "Will there be a Splash 3?" That question has always been answered in the negative. There have never been plans to do a third-version system, largely because Splash 2 is, if anything, already too complex and contains too many features.

This is not a statement that Splash 2 is flawed in its design, but rather the simple admission that it would make a poor "product" in its present form, something that has been recognized by the commercial licensees—none of the commercial versions contains all the features of the original Splash 2 system. Splash 2 was designed to be large enough to deliver high performance through parallelism, and yet few applications really used anything like the full complement of hardware that could be assembled. It was designed with a rich interconnect structure, and yet many applications use only a small part of the interconnect.

In general, we find that while all the features of the system have been used at one time or another, any single application uses only a subset of the features. And, given that we are on the edge of what can reasonably be put on a board or in a system, the cost of the features is not linear with their number. If we had to do it all over again, there are certainly some things we would change. With more pins on a Xilinx chip, we could have a 32-bit data path to memory instead of only 16. With the newer, larger, Xilinx FPGAs, we could get more logic on a chip and board and achieve higher performance. We have an extra address pin left over, and we would certainly like to double the memory attached to each FPGA. But these possibilities, intriguing as they are, represent incremental changes in the hardware to the inevitable progress of technology. What should concern us more is not the moving target of state-of-the-art technology but the broader choices of architecture, programming style, and applications for which a Custom Computing Machine makes sense.

It is within this broader framework that we realize that no good follow-on to Splash 2 exists because the major goals of the research effort have been met. Splash 2 was largely a research prototype, although some of the requirements for "real work" to be done go beyond those normally expected of such a prototype. The major goals were to build the attached processor, to demonstrate its computational effectiveness, and to demonstrate that it could be programmed. These have been met, and although there are many research questions to be addressed, none of these require the building of a "bigger and better" next version of *this* machine.

This is not to say that Splash 2 is "the last word" in CCMs. Rather, it is to say that the benefits to be gained from building a Splash 2-like machine for *research* purposes probably do not outweigh the costs. If one had real applications and real customers for a similar machine, the conclusion on costs and benefits might be different, but the decision for research purposes seems clear. A bigger machine does not seem warranted. Splash 2 was extensible in terms of number of Array Boards beyond what we found we had applications to support, and although one could now build, with flat-pack FPGAs, a board with more compute power on it, it does not seem clear that research conclusions could be drawn from the new system that could not be drawn by extrapolation from Splash 2.

The Array Board architecture similarly seems, if not optimal, at least sufficiently general yet capable of high performance, such that variations within its genre are unjustified. The two basic modes of data flow—linear and broadcast—are well supported and augmented by a crossbar whose full range of capabilities was never needed. Here, as elsewhere, we believe the research value of this part of the design space has been adequately explored. We can easily imagine a worthwhile machine produced for a niche market that resembles Splash 2. We can easily imagine other architectures (a richer hierarchical machine, for example, with clusters of FPGAs at each level of a tree structure). We can easily imagine that changes in or improvements to FPGA technology (for example, greater on-board memory, perhaps content-addressable memory, incorporation of higher-level functions, incorporation of FPGAs onto multi-chip modules) might introduce new reasons to engineer a Splash 2-like machine. But absent these justifications, we do not feel that research conducted in the building of another Splash 2-like machine is likely to lead to conclusions that could not readily be predicted from studies on Splash 2.

It is worth mentioning one major architectural feature that one would want to change in a next-generation machine. Splash 2 was oriented toward computations in which the data streamed past processing elements. The 36-bit-wide data path allows both parallel single-bit streams or wider, word-oriented streams. On a given Array Board, substantial interconnect allows for adjustments in time of the data stream. Similarly, when programmed as a SIMD machine, extensive broadcast capability exists, as well as an efficient back door for removal of a result stream. Looked at this way, the next architectural step is obvious, and almost impossible. One would like to provide, at a board-to-board level, the rich interconnect that exists on the individual boards. This is the problem we dealt with in Chapter 4 in discussing the evolution of the Splash 2 architecture. Providing the same level of interconnect among the boards that the FPGAs have on each board is a complicated matter, however, and one must ask whether the payoff justifies the expense. The answer, in terms of good applications that were made impossible due only to insufficient board-to-board communication, is no.

The problem of board-to-board communication is not unique to Splash 2 and its orientation toward a linear data stream. The DEC PeRLe PAM, with its Xilinx chips arranged in a two-dimensional grid, suffers from the same problem—at some point, an application might outgrow a single board and require substantial communication from one board to another. Fortunately, however, with Splash 2 we seem to win on both fronts. Not only does it appear that most reasonable computations can be done with at most a small number of boards requiring little communication among them (and we are sincere in our belief that we have not begged the question here), the omnipresent march of technology makes it possible to put more and more onto a single board, so that the problem should be getting less, rather than more, pronounced with time.

In retrospect, the most problematic feature of the Splash 2 architecture—the crossbar[1]—was perhaps not worth the effort, although there was no way to predict the events that occurred. The features of the crossbar—multiple configurations, dynamic choice of configuration, one-tick latency—were all used in one application or another, but each can be obtained (at some cost) by means of other switch chips or architectures.

12.3 IF NOT SPLASH 3, THEN WHAT?

Having decided that Splash 3 is not in the offing, it is reasonable to ask what sort of future research does make sense. We do not feel that the end of the CCM idea has been reached, and we expect that, in addition to other machines independently designed, several variations on our general theme (Splash 2α, Splash 2β, ..., if you will) will appear.

What we have claimed in the previous section is that the Splash 2 line of research machines for demonstration purposes is (at least temporarily) at an end, with strongly positive conclusions: sufficient compute power exists in an attached processor to obtain high performance, data can be delivered at a rate high enough to keep the processor busy and meet real-world constraints, and the machine can in fact

[1]The reader should consult Appendix A for the saga of the crossbar chips.

be programmed. What we see as future work is an elaboration of the hardware and software ideas for CCMs, now that we know that such elaboration could be worthwhile: not only must existence precede essence, but in the real world of engineering design the existence of follow-on machines must be justified by the success of their predecessors. We present some thoughts, then, on areas ripe for further work.

12.3.1 Architectures

There are strong arguments in favor of a trend toward physically smaller rather than larger systems. It is difficult to justify the price of very large systems, and such systems, with the added cabinet, backplane, interfacing, and such, are inherently more cumbersome to build. Also, as systems get physically larger, it becomes more difficult to keep propagation delays down. CCMs tend to get much of their performance advantage from tightly pipelined and carefully, explicitly, synchronized computations; these become more difficult to achieve in a system in which the propagation delays, which must be taken into account, have more than one value.

Mitigating the problem of justifying large systems is the fact that as technology advances, small systems tend more and more to deliver the processing capability of large systems. A further advantage that comes with making systems smaller and therefore cheaper is that they can be specialized to a particular collection of applications. These CCMs are inherently things that need not be single-purpose but are not likely in the near future to be general-purpose; one clear trend is toward programmable systems within a particular market. For example, there have been several designs from commercial vendors that combine DSP chips and FPGAs on a single board, aimed at signal-processing tasks of various kinds. None of these of which we are aware are "programmable" yet in the sense that Splash 2 is programmable—applications are still designed using CAD tools. But with the success of Splash 2 and of the DEC PeRLe systems and the growing awareness of the ability to make detailed circuit design unnecessary by the use of higher-level tools, we have no doubt that programming of such systems will come in the near future.

If physically smaller systems seem to be the trend, the following is, we feel, an argument against *logically* smaller systems. For the foreseeable future, CCMs will be one to two generations behind general-purpose machines, since commodity microprocessors and not FPGAs drive the technology and the market. In terms of logic performance (that is, clock rate), general-purpose machines start with about an order of magnitude advantage over CCMs. A CCM must overcome this disadvantage just to break even. Then, in order to cover the additional costs of hardware and software, download time, and such, one can argue that the CCM needs another order of magnitude in performance improvement to be considered a serious competitor. These performance advantages are presumably to be made up through parallelism in the application running on the CCM, but how small can one make a CCM and still obtain at least 100-fold parallelism? For the next several years at least, we would argue that systems with only a small number of FPGAs simply will not have the compute power to be competitive.

Although it is not technically our province to comment on the architecture of FPGAs themselves, at this point we discuss aspects of chip architecture that directly affect their use in CCMs. In this discussion, although two competing themes emerge, we do have a preference. At the grossest level and with the greatest of oversimplifications, there are two extant architectures in FPGAs. A coarse-grain architecture has 2-bit or 4-bit logic blocks and routing resources around the blocks. A fine-grain architecture, by contrast, has 1-bit logic blocks (lookup tables with two or sometimes three inputs, but only one output and only one stored value) with routing of lines going through the blocks (and thus making them unavailable for other purposes).

On the one hand, the larger logic block of the coarse-grain architecture is attractive, and the 4-bit block especially so, for the purpose of doing arithmetic. On the other hand, the routing of signals in the fine-grain architecture design is "local," so that portions of the chip can be identified with portions of the design. If an ultimate goal is to dynamically change part of the design on a chip, the fine-grain architecture is preferable. It avoids one of the problems of the Xilinx architecture, which is that the signals on routing resources adjacent to CLBs can come from distant parts of the design and be relatively unrelated to the computation being performed in the CLB.

We have already mentioned the issue of including memory (in quantity) with the routing and logic on an FPGA. This would allow the processor element/memory pairs of a CCM to be shrunk onto the FPGA itself (or even multiple replicated processor/memory pairs on a single FPGA).

12.3.2 Custom Processors

We have said nothing for the most part about one of the most enticing uses of FPGAs for Custom Computing Machines—the idea of a custom coprocessor or customizable processor. If one traces the development of microprocessor architecture through the 1970s and 1980s, one can find arguments both for and against the inclusion of coprocessors in modern workstations. Long ago, in the heyday of such chips as the 8086, math coprocessors also flourished to do the arithmetic functions that just would not fit on chips of that era. Now we find in most modern high-performance workstations both floating-point and integer arithmetic in the processor chip, and 64-bit arithmetic at that. One can legitimately argue that any further "special functions" that might benefit from an FPGA coprocessor are probably things that could be included in the next generation's processor as a matter of course.

On the other hand, it is probably true that among all the computations performed that need high performance, a rather broad range exists of "special functions" that would be desirable to have as processor instructions and not in software emulations. Whether any one of them would be deemed significant enough to warrant its inclusion in silicon is questionable, and the full list of such possible instructions is no doubt much longer than what would be feasible in the near future. A more interesting—and feasible—idea is that the FPGA resource could be incorporated directly onto the processor chip. If the math coprocessor can make the move, why not the customizable processor?

A further argument against coprocessors is the extent to which the low-level hardware and software of the machine must be adapted to permit the coprocessor to be used. In order for a coprocessor to be of value (implementing an instruction not found on the processor, for example, just as the 8087 implemented arithmetic not present on the 8086), the connection between processor and coprocessor must be very tight. Control of execution of the processor and coprocessor must be maintained and data passed between the two with the barest minimum of overhead. Exceptional conditions probably need to be handled in hardware. Most important, it must be possible for the compilers and the operating system to recognize when use of the coprocessor is advantageous, to arrange in advance for the coprocessor's "program" to be loaded, and to handle use of the coprocessor so that the only way a user would detect its presence would be by the decreased execution time.

If the future of coprocessors seems uncertain, the future of genuinely customizable processors seems less so. The dbC approach seems to go the old Burroughs B1700 one better than its multiple instruction set architectures. However one designs an Instruction Set Architecture, the fact will remain that much of the silicon resource on a chip is not actually in use in any given clock cycle. An advantage of the dbC approach is that, at least as far as the individual program is concerned, only those resources that are needed must be included. When the day comes that an FPGA (or its technological successor) permits dynamic reconfiguration while in execution, one could envision swapping portions of "processor hardware" in and out as needed. A more limited silicon resource would provide more capability by being reusable for multiple purposes.

The key to the above idea must come in the ability of the compiler and operating system to identify "processing units" and locality thereof, to extract and synthesize these units, and to manipulate their caching and loading with the same facility that virtual memory is handled today. And this idea will probably not be relevant to all forms of computation. There is and will no doubt continue to be a solid market for machines that do those things we now consider ordinary, and unless there is a substantial portion of a computation that is simply not done well on a traditional machine, there will be no incentive to try a reconfigurable processor—custom silicon will always be faster, and mass-market commodity machines will always be cheaper. But the quicker time-to-market of programmable hardware is an advantage, and if a selected set of niche markets were to be determined and were then targeted by commercial operations capable of carrying out successful business plans, then we feel that such reconfigurable processor machines, whose underlying processor architecture was defined only at compile-time or runtime, could become almost commonplace.

12.3.3 Languages

Without doubt, the deepest and most fascinating question regarding the evolution of CCMs is that of their programming models and languages. This is the thorny issue that has bedeviled those responsible for language software for parallel computers for nearly two decades. How much detail of the machine should the programmer see? What is the penalty in performance for a high-level view? Users of high performance computer systems have usually been willing to endure in the name of speed some agony not suffered by those for whom speed is not so vital, but it is also true that there is a limit to the patience of even these stalwarts. Should the cost of programming surpass an ill-defined and yet very real threshold, the cost is not merely an incremental loss in the number of users and applications but a rejection of the entire system.

We can look to several different experiences for insight in this issue. Most significant is our own experience with VHDL and Splash 2. After that, of course, we can make comparisons with dbC on Splash 2. Finally, there is the work of others, such as the C extension done at DEC Paris and the VHDL work done by Box at Lockheed Sanders for CHAMP [1]. All of these can also be viewed merely as the first steps taken, in part because one could capitalize on existing knowledge and tools. A necessary further step is to contemplate in the abstract what would be most desirable if one were free of the need to consider present cost, personnel, past history, and backward compatibility.

A great many of the Splash 2 applications were done not as procedural programs but as a series of processes pipelined together, through which data flowed synchronously. These resemble nothing quite so much as programs in discrete event system simulation, and a language like VHDL seems highly suitable for this kind of programming. The `SIGNAL` data type provides for explicitly concurrent events and allows the programmer to express in a natural way the parallelism inherent in a computation. The fact that `SIGNAL`s are updated with every clock tick allows the programmer to specify very precisely what the synchronization of the concurrent processes is to be. The alternative of the `VARIABLE` data type, by contrast, is suitable for procedural segments of code or for code over whose execution the programmer need not take such care.

The negative side of the program control offered by the explicit parallelism of `SIGNAL`s in VHDL is that the programmer *must* in fact synchronize the updates and that "off by one" errors in choreographing this process can be common. We feel that this does not argue against VHDL so much as it argues in favor of spreadsheet-like tools that facilitate such programming. The expressiveness of a genuine parallel language (which VHDL most certainly is) seems to be necessary to achieve the needed performance. Rather than abandoning the parallelism because it can make programming difficult, one must work to compensate for the difficulty, with better tools.

If many of the Splash 2 applications resemble discrete event system simulation programs, they are also like systolic programs or data flow programs. They differ from the former in that the processes can vary widely in type and size and the programs are not nearly so well-structured as are systolic programs. And they differ from data flow programs in that they have *more* structure—the expected performance advantage comes in part from the tight pipelining and synchronization of the processes, as mentioned above.

We contemplated at several points in the Splash 2 project an investigation of one or more of the various languages available for programming in which the control of execution comes not from an instruction sequencer but from the synchronous flow of the data. We have no doubt that for many applications this might be a much more natural model of computation than presently exists. That we have done no such investigation is due entirely to the fact that we had to stay focused on the main goals and could not allow ourselves to be too distracted by curiosity from those ends which had to be accomplished. In the eventual fullness of time, however, we expect that such a study would be of great value.

One major drawback to the use of a data-driven language and model of computation must be raised. Programming of Splash 2 in VHDL has already proved to be a bit of a hard sell because VHDL "just isn't C." VHDL is nonetheless a DoD standard, taught to students across the world, used in industry, and supported by very sophisticated software tools. With all this in its favor, and working against it only the religious objections and the concerned hand-wringing of middle managers whose performance appraisals depend on quantity of present output, how much harder would it be to gain acceptance of another language, which no doubt would be viewed as even more exotic?

It was to a great extent in response to the above concerns that we discussed augmenting the standard VHDL framework with features that would make programming Splash 2 much more C-like (a VHDL++, as it were), or in the other direction removing from the available VHDL language tools those aspects not needed for Splash 2 programming and potentially confusing or threatening to applications programmers (to produce VHDL−−?). Some of each would seem desirable.

We remark finally that with two different applications the price paid both in FPGA resources used and in speed of execution was about a factor of three or less between handcrafted XACT designs and synthesized VHDL code in the normal Splash 2 programming model. We feel that both are acceptable. The resource estimate was with an earlier version of the synthesis tools than is presently available, and may already have improved. The speed differential is not much different from that between high-level language and assembly code, and thus is not likely to be the deciding factor, except for those few applications that are even with XACT implementations running on the margins of acceptable speed.

If the questions surrounding Splash 2 and its normal VHDL programming model are not of capability but of acceptability, then almost the opposite is true of dbC. The language here clearly *is* C, or as close to C as one can expect to get and still be running on a SIMD machine. There are two basic questions: Can the performance be great enough to be adequate? Is the range of SIMD applications broad enough to justify the use of a different language for them?

We have remarked on the factor-of-three performance difference between XACT and VHDL. It has been further noticed that roughly the same difference exists between dbC programs and their "directly VHDL" counterparts. This comes to a factor of nearly an order of magnitude, which is probably not tolerable. (The genome sequence comparisons mentioned in Chapters 8 and 9, in contrast, show a factor of 150 superiority for the VHDL version.) There will no doubt always be some penalty for generating the code automatically through dbC; it remains to be seen whether the minimized value of this penalty is small enough.

We are much more sanguine about the breadth of SIMD applications. There are several computational problems—including much of image processing, one natural area for CCMs—that can be done very effectively as SIMD computations. An additional argument in favor of a programming model like that of dbC is that SIMD programs have the same sort of carefully sequenced flow of control that the Splash 2 VHDL programs do. Thus, although the applications are limited and there is a danger that one might need a VHDL-like programming model as well to handle non-SIMD aspects of even a largely SIMD computation, we expect that continued work on dbC is reasonable and will find use in real applications.

We comment finally on a matter that is not just a matter of language but of the entire programming process for CCMs, and that is the question of upward compatibility. It has been crucial in many computing environments for established users to be able to upgrade hardware without substantially changing programs that represent their investment of time in the process of solving their problems. It seems unlikely in this early stage of marketing of CCM that users will be able to avoid some level of discomfort at the changes in the hardware, programming, and logic synthesis tools underneath their applications. Clearly, then, to be successful, the benefits in improved performance will need to be able to overcome this drawback.

12.4 THE "KILLER" APPLICATIONS

It seems a staple of the computing industry's folklore that novel products like CCMs need to have at least one "killer" application for which the new product is so well suited that it is clearly the preferred choice. Once the product has gained a foothold in the commercial marketplace and can be viewed to "exist" in a serious sense, broader usage is then to be expected. This is part of the very real spinoff and serendipity side of technology advancement.

What, then, might be those killer applications? Three broad categories seem clear: a) image processing; b) real-time data handling and control, in which one finds large volumes of data with computations that are limited in complexity but relatively unusual if done on standard microprocessors; and c) rapid prototyping and architecture emulation, in which reconfigurability of a platform is essential to allow exploration of alternatives, but for which some sort of hardware solution is required to provide answers in a reasonably timely manner.

The two chapters on video processing and fingerprint matching are illustrative of the first of these three categories. The number of basic operations to be performed in unit time is very high. The operations themselves are not "standard," often because arithmetic using relatively few bits is possible. There is a high degree of parallelism and/or pipelining in the modest collection of algorithms that need to be implemented. These argue in favor of a hardware solution. And, arguing against ASIC development, the computations or data formats are not so totally standard and structured that today's full-custom hardware can be expected to provide a longer-term solution. Arguing further in favor of a CCM is that while hardware can be built to handle data or image compression, convolutional filtering, signal encoding or decoding, and so forth, with the use of reconfigurable hardware one can use the same hardware, or at least replicated versions of the same hardware running different programs, rather than requiring multiple distinct parts. The obvious advantages then apply with respect to building and maintaining the hardware and the application programmer/designer being able to implement and maintain programs on the final system.

Perhaps the best present example of real-time data handling or control using a CCM is the use by Moll et al. [2] of the DEC PeRLe-1 system in handling data from experiments to be run on the Large Hadron Collider soon to be built at CERN (the European Organization for Nuclear Research, Geneva, Switzerland). The plan is to use PeRLe in the second of three levels of data filtering before the data is saved off for further study. Here, there is a need for a flexible or reconfigurable processor and for high-performance processing in which substantial parallelism exists, and the data flow rate is high. A link from the PeRLe host TURBOchannel to HiPPI will provide the high data rate (a similar HiPPI-to-Splash 2 interface went through early design at SRC but was never completed for lack of a good target application or system that would use it). The flexibility of a CCM is an asset here in part because this is an experimental framework—unlike the day-to-day handling of large volumes of data that might take place in a commercial environment, one can expect the requirements at CERN to change over time with different experiments and different variations of the same experiment.

Very little has been said in this book about rapid prototyping using a CCM. This is due to our concentration with Splash 2 not on its use as an engineering tool but as a machine to be used for computing. But the use of FPGAs for prototyping is already well developed, as is evidenced by the health of companies such as Quickturn. The Quickturn hardware is geared, however, toward circuit design rather than system design. Emerging from several ongoing university projects, however, is the ability to test component-level issues rather than chip-level issues—processor interactions with memory, various memory and caching schemes, bus strategies, and such. Splash 2, for example, is presently being used to study a proposed parallel computer architecture. We suspect that, as good as hardware such as that from Quickturn is for many of the design uses to which it is put, it may not work well on higher-level architectural emulation, and that what will be needed is a system of the nature of Splash 2 with its built-in data path, explicit connections to memory, and so forth. The basic boxes of a computer architecture's block diagram are already present in Splash 2; they're just somewhat more amorphous than in a "real" computer.

Although the emulation on Splash 2 of a proposed architecture would be slower that the hardware itself, the parallelism of the machine can make it much faster than software simulation. Importantly, although one could not expect a proposed architecture to map directly to Splash 2, the partial structure of Splash 2's data and memory paths and its processor interconnections would allow many architectural features that did not fit directly to be time-multiplexed in a measurable way that would permit accurate extrapolations.

12.5 FINAL WORDS

We close this book with the not-very-bold statement that we doubt that these will be the last words spoken about Custom Computing Machines. We hope that what we have produced is more than just a project report and that a study of our system, taken as a whole, can provide insight to others planning related work. We believe we have influenced the course of research in CCMs by what we have already done, and we hope that somewhere in these pages will have been found a satisfactory explanation of the paths we took and the choices we made.

REFERENCES

[1] B. Box, "Field Programmable Gate Array Based Reconfigurable Preprocessor," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 40–49.

[2] L. Moll, J. Vuillemin, and P. Boucard, "High-Energy Physics on DEC PeRLe-1 Programmable Active Memory," *Proc. FPGA95*, ACM, ACM Press, New York, 1995, pp. 47–52.

[3] M. Shand, P. Bertin, and J. Vuillemin, "Hardware Speedups for Long Integer Multiplication," *Proc. ACM Symp. Parallel Algorithms and Architectures*, ACM, ACM Press, New York, 1990, pp. 138–145.

APPENDIX A

Splash 2 Development—The Project Manager's Summary

Duncan A. Buell

I fully admit now that when Fred More first approached me in the summer of 1991 with the idea of my supervising the general development of a second version of the Splash processor, I had no idea what I was getting into. I certainly didn't expect this to turn into a virtually full-time job for two and a half years, or else I might well have said no to the idea. In hindsight, it is clear that my ignorance was a good thing, for I think that Splash 2 was a solid success as well as the most exciting piece of work in which I've had the chance to be involved.

After some serious thinking about the issues, I told Fred I'd do it. I had been very interested in the first Splash machine, but had been unable, due to other pressures, to do direct work on it. My line management position had left me with very little time to work directly on research projects, and in every instance in which I had found time, I hadn't found enough time. I had wanted in one instance to write a program that was essentially a double loop with a table lookup in the body of the inner loop. After an entire afternoon spent trying—and failing—to construct the counter for the outer loop, I gave up. Although there were a number of people who had programmed Splash with great success, I was unlikely to become one among them.

The task that Fred More originally offered me was to rectify the problem that led to my frustrating admission of failure. The hardware of Splash was a solid success; it ran as expected and had few, if any, failures. Similarly, the software was as good as one could hope for, given the time and context. Maya Gokhale's LDG had been a tremendous advance for the intended purposes over the still-developing Xilinx tools. But the problem remained that it was an FPGA-based machine on which one could design circuitry to perform applications, but not a machine on which one could *program* applications. Fred's charge to me was to drive the development from the applications perspective and from the point of view of an applications programmer. The goal was to be able to say that programmers without a background in hardware design could write applications and achieve moderately high performance.

In accepting Fred's offer, I had one condition—I was perfectly happy to mount a search for applications that could perform well on this machine and to deal with the problem of getting the sense of "programming" into Splash 2, but I insisted that I have someone working with me who would feel responsible for the actual hardware development and someone else who would do the same for the systems software. The hardware person was to have been Andy Kopser, until he announced in late summer that he would be leaving SRC in mid-September. Elaine Davis took over the hardware, to be succeeded by Wally Kleinfelder when Elaine left for a new job the following February. The software position remained unfilled until late October, when Jeff Arnold agreed to take on the job.

From the very beginning, it was assumed that building a small system and programming kernels as benchmarks would be insufficient justification for claims of success. It would be necessary to have a system large enough to do, if not real problems, at least problems of a size comparable to real ones.

In terms of the scope and nature of the applications programming process, my agreement with SRC management was the following: We would make an honest attempt at perhaps a dozen problems. Three to six of these would be genuinely unsuccessful, either because the problems would fairly quickly be found to have a show-stopping component for Splash 2 or because the projected payoff would appear too low to warrant a complete experiment. Of the six remaining, half would prove to be successful "experiments" with nonpositive results. That is, the experiments would be complete enough that hard performance numbers could be obtained and an objective analysis of results made, but the results themselves would not show that Splash 2 was a big win or a win big enough to warrant for "production computation" the use of unusual extra hardware and the attendant problems of programming, interfacing, and maintenance. Finally, it was assumed that perhaps three of the original dozen applications would prove to be major successes, and that this would be sufficient to declare victory for Splash 2.

In order to obtain the dozen attempts at Splash 2 applications, I asked for and received from management at SRC "12 applications" worth of people, figuring the unsuccessful six applications at one to three months' effort and the successful six at three to six months' effort. Looking back, I believe that little or no revision is necessary to assert that this was, in fact, the way things went.

Funding for Splash 2 came from a special DoD "dual-use" appropriation. On October 16, 1991, an SRC presentation was one of about 35 made to various civilian government agency representatives. The requirement to obtain funding was not only that the project be technically worthy of funding; it was also necessary that some civilian agency sign on to be a recipient of the technology transfer. In our case, the recipient was the Department of Mathematical Biology of the National Cancer Institute (NCI). From the very beginning, we had contracted for delivery of a Splash 2 system and working code for the sequence comparison problem as part of a Memorandum of Understanding with NCI. Although final funding approval did not come until the spring, we had the go-ahead, rather shortly after the October 16 presentation, to continue with Splash 2.

Architectural design proceeded through the fall of 1991. The actual engineering and construction of Splash 2 were to have been done under a contract with a private company that was handling both Splash 2 and another novel machine—TERASYS—being built by SRC. TERASYS had started about four months earlier than Splash 2, and the first change in the general project plan came in February of 1992. Due to cost overruns, it was clear that TERASYS and Splash 2 could not both be completed under the outside contract. Since TERASYS was nearer completion, a decision was made to pull back into SRC the design and construction of Splash 2.

This was to be the first of several headaches. An ongoing problem was that of obtaining cabinets in which to house the Splash 2 system. The early choice of the Futurebus+ backplane by the contractor proved to be ill-advised. We went through no fewer than three complete bid procedures to obtain cabinets and backplanes—vendor A supplied one model A cabinet, then got out of the Futurebus+ business; vendor B then did the same thing, by the end of which time vendor A was back in the Futurebus+ business and supplied still a third version. Fortunately, all models actually did work, but the lack of uniformity and the effort spent in procurement was a great annoyance.

A more critical problem was the discovery, in the spring of 1992, that the TI switch chips planned for use in the crossbar were no longer in production. By this time, we had committed to a planned 10 Splash 2 systems, some 40 array boards, needing a total of 360 switch chips (plus spares). We quickly cornered the market on the switch chips known to exist, although we were naturally forced to pay a premium price for them. From then on, the number of available switch chips was the limiting factor in the number of Splash 2 systems that could be built. Later, when technology transfer was being discussed with commercial enterprises, this was the single greatest sticking point, which more than once almost brought things grinding to a halt.

Had we been able to change switch chips, even at that relatively late date, we might well have done so, but there was not then and there still does not exist a genuine substitute for the TI chips we had chosen. We felt we needed on the array card the ability to get across the crossbar in one tick and the ability to change, on a tick-by-tick basis, the configuration of the crossbar. The former capability allows a programmer to treat crossbar or linear FPGA-to-FPGA data transfers as identical, so that algorithms and programming do not require explicit pipelines or hierarchy. The latter allows flexibility in an algorithm and reduces the impact of a scarcity of resources.

Later revisionist thoughts on how the crossbar should or could have been done included using FPGAs or multiple Aptix chips. The TI chips permitted as many as eight configurations, but no applications that were implemented actually used more than four. The longer time for reconfiguration required by either alternative could have been taken care of by having as many as four devices on a board and the choice of configuration made with a multiplexor selection of one of the four "static" options.

Although progress on the Splash 2 hardware seemed at times to go in fits and starts, progress on the software was rapid through 1992. By the end of February, a working version of a simulator for the Splash 2 hardware existed, and a brief workshop was held at the end of the month to train the first guinea-pig group of programmers. Although one of the initial group was so disillusioned as to vow never to get near Splash 2 again, the general response was guardedly positive. We were indeed in uncharted waters, using for programming a language (VHDL) not intended for programming, and using VHDL tools from Synopsys and Model Technologies for purposes other than those for which they had been intended, by users much more naive (with regard to circuit design) than was ever the plan of these vendors. Also, the simulator was imperfect and incomplete at first.

All in all, programming the Splash 2 simulator in the spring and early summer of 1992 was not an entirely pleasant task. But we had begun the project with only a hazy understanding of what we needed and wanted, and it was crucial to the development of the software environment that genuine efforts be made to use the tools. We could not have laid out the specifications *a priori*; what evolved was a compromise between what was needed by the programmers and what was possible given the tools.

The patience and cooperation of the "programmers" in this period was matched only by the skill of those who were continually rewriting and upgrading the simulator and tools, notably Jeff Arnold. In a world of modern windows-based software tools, we were necessarily conducting a human-factors experiment on our own people on the level of frustration acceptable to goal-oriented application programmers working with changing tools. Remarkably, with the one exception, all the commitments were carried through to completion, and the systems software personnel for their part survived the onslaught of users clamoring for bug fixes and the instant implementation of the planned features currently holding up their progress.

Beginning with discussions with Synopsys management in June of 1992, we attempted to influence the development of VHDL simulation and synthesis tools aimed at "programming" applications on (to begin with, Xilinx) FPGAs. This led to a contract with Synopsys for a product later to become their FPGA Compiler. For several months Jeff Arnold went back and forth with Synopsys on a list of needs and wants that would make their tool look to a programmer more like a "regular compiler" for a language like C or Fortran.

Crucial to eventual success was the discovery during this period that the underlying Xilinx hardware and software was a significant improvement over what had gone before. Two problems with the XC3090 chips and their attendant `apr` software for placement and routing were that the chips themselves were a little too small to accommodate a natural "unit of computation" for many applications and that `apr`, as it existed in about 1990, had major drawbacks. It often either took too long to run or failed to route an entire design, especially if left to work "automatically," that is, without human intervention to guide the placement and routing. We intended to use the XC4010 `ppr` software as "automatic" software without any help from a user assumed to be uninterested or unable to help the design process. It was a great relief, then, to find that it was possible to write VHDL code for realistic applications that used a significant fraction (75 percent or more) of the XC4010 chips and to have the Synopsys and Xilinx tools synthesize, place, and route the program/design into a Xilinx bitfile that would allegedly execute at 10–25 MHz. One reason for this improved performance of the placement and routing software clearly seemed to be that the XC4000 series chips have a much better balance between logic resources and routing resources. In one very special instance of the DNA sequence comparison program, which has an extremely regular structure, it was possible to utilize all of the CLBs on a chip and still have the VHDL-to-bitfile translation take place without human intervention.

The summer of 1992 was a vigorous and rowdy period in the life of the Splash 2 project, in part due to the presence of five summer students working on various aspects of hardware, software, and applications programs. It was in this period that the explosive growth toward a usable programming environment took place—a large number of both small and large applications were tried, fixes or work-arounds for problems or bugs were found and shared, and tools to assist program development were written. (As always, the work of programming benefits from the deep and abiding sloth of students who insist on writing tools because they are too lazy to do things "the hard way.")

From the beginning of the project, and continuing through until about March of 1993, we had been conducting a vigorous search for good test applications. Over the course of the project we spoke at more than 20 universities, 15 companies, 10 government agencies, and 9 conferences. From the very beginning, of course, we had the sequence comparison problem from NCI as a "must do" application, and work began early in 1992 on a solution to this problem, leading to the paper presented as a later chapter of this book. But this by itself would clearly not be enough.

A potential problem from the National Center for Biotechnology Information involving clustering of bibliographic records was a moderately good match for Splash 2, but a single complete run would take two years (compared with 10 months on a Thinking Machines CM-2 supercomputer); this was dropped. Discussions with a NASA contractor on the use of Splash 2 as a platform on which to do rapid prototyping were positive. An engineer from the company spent several weeks at SRC and came away with very positive thoughts, but the lack of extant hardware to borrow or buy was probably the show stopper in that deadline-driven world of government contracting—we were a little too far ahead of the curve for them to use Splash 2 to advantage.

I had visited VPI, however, on an early speaking trip through North Carolina and Virginia in September of 1991, and our discussions with members of the Electrical Engineering Department had continued. Peter Athanas had recently finished his Ph.D. at Brown University working on the PRISM FPGA project, and had then joined the faculty at VPI. Lynn Abbott had interesting problems in image processing and a desire to explore the use of FPGAs in hardware to accelerate the computations. The presence of Jim Armstrong suggested strong support for and solid expertise in VHDL among the students. All this was helped by the fact that John McHenry, who had spent two summers at SRC working first on Splash 1 and then on Splash 2, was finishing his Ph.D. in the department and knew the program well. When IDA Headquarters made money available for a university contract for Splash 2 applications and research, VPI was a natural choice. The ongoing relationship has been close and profitable, and a summary of their work on Splash 2 appears earlier in this book.

The variation of image processing necessary to do fingerprint matching had been discussed as a possible Splash 2 application at the October 16, 1991, presentation to government agencies. We had, at times, talked with the FBI and with potential government contractors about machines to match fingerprints, but had failed to land upon a definable experiment that could be performed. The second IDA contract thus went to Anil Jain and Diane Rover at Michigan State University after a trip I made there in January of 1993, and their work is also reported here.

Not everything was proceeding smoothly, however. What with having to pull the design back from the contractor and the switch chip and cabinet/backplane delays, a planned "hardware working" date for fall of 1992 slipped, then slipped again, then slipped still further as some of the Splash 2 engineers were time-sliced with the ongoing TERASYS project. Such delays might have killed Splash 2 in an organization that required a marketable product or needed to keep tighter control on employee time invested. At SRC, however, although we were always subject to the possibility that key players would feel compelled to drop everything to take advantage of a window of opportunity elsewhere, we were allowed to make our steady but sometimes slow progress. One could even argue (if one had to) that the hardware delays worked to the benefit of the system results by allowing more time to be spent on debugging and streamlining the process of developing applications code.

Finally, on Thursday afternoon, February 18, 1993, the first Splash 2 hardware system worked. Jeff Arnold downloaded an edge-detection program to an array board, streamed the pixels of a digitized image through the board, and received as output the edges of the image.

From then on, replication of the hardware components was rapid. Although we had stretched our resources to the limit in committing to build 10 Splash 2 systems, demand soon exceeded the supply. In addition to the systems committed to VPI, MSU, NCI, and to SRC for its own purposes, university researchers and outside companies were beginning to call to ask how copies of Splash 2 could be bought or borrowed. Even without the obvious problem presented by the switch chip, SRC was faced with a very difficult dilemma. Further Splash 2 clones were impossible without either redesigning the array board (and modifying the systems software accordingly) or designing a new, functionally equivalent and pin-compatible chip to fit into the existing board design. Neither option seemed attractive. Further, it was clear that SRC could not afford the real dollar and personnel cost of becoming the manufacturing and maintenance organization for Splash 2. Success, in this case, could come with a heavy price tag.

After some months of deliberation and at least one false start, SRC's government sponsors undertook in the first part of 1994 the technology transfer and commercialization of Splash 2. Outside private companies were to be granted, for $100, a complete data dump of schematics, diagnostics, manuals, and code, together with some guidance from SRC about things done right, things done wrong, and things that should simply be done in a different way. An initial group of potential licensees was brought to SRC for a presentation in March of 1994. The first license was issued soon after that; within six months 10 companies had obtained licenses, and by the end of calendar year 1994 the first commercial Splash 2 derivatives had become available.

Not all of the many licensees have the intention of producing anything like a commercial version of Splash 2. There are several other processors, board, and systems available or nearly available from other companies; some of the licensees have more of an interest in the software we developed for programming an FPGA-based machine or the general systems approach we took than in the specific details of Splash 2. A small consortium of licensees has formed to target an image processing market; the companies involved have divided the hardware, software, and applications into areas where each can contribute from its strength and benefit from cooperation with the others.

From the earliest days of the Splash 2 project, I had insisted that we could declare victory if any one of these criteria were met:

1. Splash 2 would be used to get real work done and not just provide demonstrations of capability.
2. Someone who did not get an SRC paycheck would use Splash 2 in his/her work and not walk away vowing "Never again!"
3. Some commercial machine would appear and have a clear and traceable ancestry to Splash 2.

It is perhaps too soon, and we are perhaps too close to the matter, to judge exactly why we succeeded; I leave such analysis to others. Having a brilliant and dedicated technical team was a major factor. Not having a particular target application helped—we were free to search for reasonable applications that would be successes. Not having marketability and "productizing" constraints helped. Not having a deadline that forced us to abandon "the right thing to do" in order to meet the deadline helped. But as it has turned out, not just one but all three of my criteria have been met. Further, we have effected something often talked about but seemingly rarely ever done—we have been able to convert proof-of-concept technology, developed at government expense as an engineering research project, into products available for sale from private-sector companies whose personnel rosters do not intersect the list of principals from the research project.

APPENDIX B

An Example Application

Jeffrey M. Arnold[1]

In this appendix we illustrate the Splash 2 programming style through an example application written in VHDL. This example, a simple digital filter, is much smaller than most Splash 2 applications, but does touch many of the issues facing the programmer. Equation 1 shows the general form of a finite impulse response (FIR) filter:

$$Q_i = \frac{\sum_j I_{i-j} F_j}{C} \tag{1}$$

where I is the input data stream, F is the set of filter coefficients, C is a constant scale factor, and Q is the output data stream.

In this example the input data is a stream of 12-bit signed integer samples, the output is a stream of 16-bit signed integers, and the filter is a five-tap low-pass filter with constant coefficients. The filter can be viewed as a weighted sum computed on a sliding window of the input data followed by the application of a constant scale factor. A block diagram of this interpretation of the filter is shown in Figure B.1.

This application is simple enough to be mapped entirely onto a single Processing Element, obviating the need to partition across multiple FPGAs. The input data arrives on the left port of the PE at the rate of one 12-bit sample per clock cycle conditioned by a valid data tag. The output data is produced at the same rate on the right port. For the sake of simplicity we assume the filter coefficients are powers of two, $F = \{1, 4, 8, 4, 1\}$, eliminating the need for combinational multipliers. The five-input add operator is implemented as a pipelined tree of two input adders. Finally, the division by the constant scale factor C is implemented by table lookup in the

[1] A version of this chapter appeared as Arnold and Buell [1] and is used with permission.

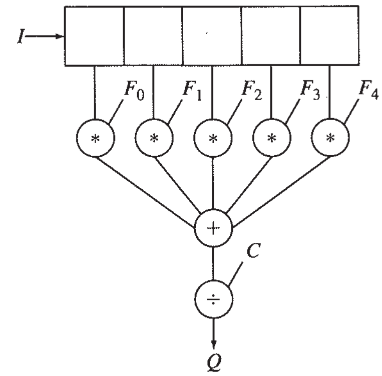


FIGURE B.1 Block Diagram of Five-Tap FIR Filter

external RAM. The output of the add operator is used to index into the table and the contents of the addressed location is returned as the output of the filter.

Figure B.2 shows the annotated VHDL FIR program. The Processing Element entity declaration is shown in Figure 6.1 and is not reproduced here. The 12-bit input stream enters the PE on the left port (`XP_Left`), a weighted sum over a five-element window of the stream is computed, the sum is scaled by table lookup in the external memory, and a 16-bit result stream is sent to the right port (`XP_Right`). The first four lines of the architecture specify data type and parametric information that would be placed more appropriately in a separate package, but are included here for brevity. Line 2 defines the type of the input stream samples to be 12-bit signed integers. Line 3 declares the data type to be used for vectors of `Samples`. The number of filter taps (the data "window") is defined to be a constant 5 in line 4. Line 5 defines the set of coefficients by which each element of the window is to be multiplied. Note that for the sake of efficiency the coefficients are chosen to be powers of two, obviating the need to synthesize combinational multipliers. In general, though, the coefficient vector could be any set of constant integers; the compilation tools will synthesize the appropriate logic.

The next five lines (6–10) are declarations of internal signal objects, the storage elements of the program. `Line_Buffer` contains the sliding window of data samples to be filtered. `sum1`, `sum2`, and `sum3` are temporary registers that hold intermediate values. The remaining signals constitute the interfaces to the external memory and to the neighboring PEs.

The body of the architecture contains two synchronous processes and one concurrent procedure call. The synchronous processes respectively compute the weighted sum and interface to the external memory. The `Filter` process declares an internal variable, `Sum`, which is used as an identifier for an intermediate value. By choosing to make `Sum` a variable rather than a signal, no register will be inferred. Within the body of the process, the call to the procedure `Pad_Input` performs type conversion from the port type `RBit3` to the intrinsic `Bit_Vector` type. By placing the procedure call within the body of the clocked process, a pipeline register is implicitly added. This is a standard practice used on most input and output ports, designed to improve performance by allowing the IOB flip-flops in the Xilinx XC4010 FPGA to be used to stage data onto and off of the PE.

The `FOR` loop in lines 17–19 shifts the data window by one sample each clock cycle. Because signal assignments take effect *after* the execution of the process, all

```
1  ARCHITECTURE FIR OF Processing_Element IS
2    SUBTYPE Sample IS Integer range -(2**11) to (2**11 - 1);
3    TYPE Sample_Vector IS Array (Integer range <>) of Sample;
4    CONSTANT NTaps           : Integer := 5;
5    CONSTANT Coeff           : Sample_Vector(0 to NTaps-1) := (1,4,8,4,1);
6    SIGNAL Line_Buffer       : Sample_Vector(0 to NTaps-1);
7    SIGNAL sum1, sum2, sum3 : Integer range -(2**14) to (2**14 - 1);
8    SIGNAL madr              : Bit_Vector(MAR_RANGE-1 downto 0);
9    SIGNAL mdata_in          : Bit_Vector(MEM_WIDTH-1 downto 0);
10   SIGNAL Left, Right       : Bit_Vector(Datapath_Width-1 downto 0);
11 BEGIN  --  FIR
12   Filter : PROCESS
13      VARIABLE Sum : Integer;
14   BEGIN
15     WAIT UNTIL XP_Clk'Event and XP_Clk = '1';
16     Pad_Input(XP_Left, Left);
17     FOR i IN 1 to NTaps-1 LOOP
18       Line_Buffer(i) <= Line_Buffer(i-1);
19     END LOOP;
20     IF (Left(35) = '1') THEN
21       Line_Buffer(0) <= Conv_Integer(Left(11 downto 0));
22     ELSE
23       Line_Buffer(0) <= 0;
24     END IF;
25     sum1 <= (Line_Buffer(0) * Coeff(0)) + (Line_Buffer(1) * Coeff(1));
26     sum2 <= (Line_Buffer(2) * Coeff(2)) + (Line_Buffer(3) * Coeff(3));
27     sum3 <= Line_Buffer(4) * Coeff(4);
28     sum  := sum1 + sum2 + sum3;
29     madr <= CONV_Unsigned(sum, MAR_Range);
30   END PROCESS Filter;
31   Mem_Access : PROCESS
32   BEGIN
33     WAIT UNTIL XP_Clk'Event and XP_Clk = '1';
34     XP_Mem_Rd_L <= '0';
35     XP_Mem_Wr_L <= '1';
36     Pad_Output (XP_Mem_A, madr);
37     Pad_Input  (XP_Mem_D, mdata_in);
38     Right(15 downto 0) <= mdata_in;
39   END PROCESS Mem_Access;
40   Pad_Output(XP_Right, Right);
41 END FIR;
```

FIGURE B.2 Body of Finite Impulse Response Program

assignments occur in parallel, so the direction of iteration is not significant. Lines 20 through 25 control the loading of the window buffer: if bit 35 of the input stream is a '1' the low-order 12 bits are converted to integer and shifted into `Line_Buffer`; otherwise a constant zero is shifted in. The weighted sum is computed in two pipeline stages by lines 25–29. In the first stage each window element is "multiplied" by its coefficient (in zero time and area, as the coefficients are powers of two), and three partial sums are computed and stored in registers (`sum1`, `sum2`, and `sum3`). In

```
28a   sum4 <= sum1 + sum2;
28b   sum5 <= sum3;
28c   sum  := sum4 + sum4;
```

FIGURE B.3 Optimized Final Addition

the second stage a three-input sum is computed, the type is converted to unsigned bit vector and zero extended to the length of `MAR_Range` (the size of the memory address), and stored in the memory address register, `madr`, in preparation for the table lookup.

The second synchronous process latches the address (the weighted sum computed by `Filter`) to the external memory, and the scaled result returned from the memory. These pipeline stages are necessary to ensure that the memory address, data, and control signals are registered in the IOBs of the FPGA. The memory control signals, `XP_Mem_Rd_L`, and `XP_Mem_Wr_L`, are held constant by lines 34–35, forcing the memory to always read. Line 38 is an additional pipeline register on the return data prior to transmission to the next PE. By registering the data here, the assignment to the `XP_Right` port may be performed outside of the process by the concurrent procedure call in line 40.

There are six total pipeline stages in this program:

- the assignment of the input data to the `Left` signal (line 16)
- the computation of the partial sums `sum1`, `sum2`, and `sum3` (lines 25–27)
- the calculation of the final sum, `madr` (lines 28–29)
- the assignment to the memory address register, `XP_Mem_A` (line 36)
- the return data from the memory, `mdata_in` (line 37)
- the assignment into the output data register, `Right` (line 38)

When this program is compiled it occupies 61 of the 400 CLBs, or 15 percent of the available real estate. The critical path delay is 106 ns, limiting the maximum clock frequency to 9.3 MHz. The static timing analyzer shows the critical path is through the three-input adder in line 28. If we needed to optimize the performance of this design further, an extra pipeline stage could be added as shown in Figure B.3.

REFERENCES

[1] J.M. Arnold and D.A. Buell, "VHDL programming on Splash 2," in W. Moore and W. Luk, eds., *More FPGAs*, Abingdon EE & CS Books, Abingdon, England, UK, 1994, pp. 182–191.

Bibliography

"Application Briefs: Computer Graphics in the Detective Business," *IEEE Computer Graphics and Applications*, Vol. 5, Apr. 1985, pp. 14–17.

A.L. Abbott et al., "Finding Lines and Building Pyramids with Splash-2," *Proc. IEEE Workshop FPGAs for Custom Computing*, CS Press, Los Alamitos, Calif., 1994, pp. 155–163.

A.L. Abbott, R.M. Haralick, and X. Zhuang, "Pipeline Architectures for Morphologic Image Analysis," *Machine Vision and Applications*, Vol. 1, No. 1, 1988, pp. 23–40.

L. Agarwal, M. Wazlowski, and S. Ghosh, "An Asynchronous Approach to Efficient Execution of Programs on Adaptive Architectures Utilizing FPGAs," *Proc. IEEE Workshop FPGAs for Custom Computing*, CS Press, Los Alamitos, Calif., 1994, pp. 101–110.

Algotronix Ltd., *The Configurable Logic Data Book*, Algotronix Ltd., Edinburgh, Scotland, UK, 1990.

R. Amerson et al., "Teramac—Configurable Custom Computing," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 32–38.

A.A. Apostolico et al., "Efficient Parallel Algorithms for String Editing and Related Problems," *SIAM J. on Computing*, Vol. 19, 1990, pp. 968–988.

J.M. Arnold, "The Splash 2 Software Environment," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 88–94.

J.M. Arnold, "The Splash 2 Software Environment," *J. of Supercomputing*, Vol. 9, 1995, pp. 277–290.

J.M. Arnold and D.A. Buell, "VHDL Programming on Splash 2," in W. Moore and W. Luk, eds., *More FPGAs*, Abingdon EE & CS Books, Abingdon, England, UK, 1994, pp. 182–191.

J.M. Arnold, D.A. Buell, and E.G. Davis, "Splash 2," *ACM Symp. Parallel Algorithms and Architectures*, ACM Press, New York, 1992, pp. 316–322.

J.M. Arnold et al., "The Splash 2 Processor and Applications," *Proc. Int'l Conf. Computer Design*, CS Press, Los Alamitos, Calif., 1993, pp. 482–485.

J.M. Arnold and M.A. McGarry, "Splash 2 Programmer's Manual," Tech. Report SRC-TR-93-107, SRC, Bowie, Md., 1993.

P.M. Athanas, "Functional Reconfigurable Architecture and Compiler for Adaptive Computing," *Proc. 1993 Int'l Phoenix Computer and Comm. Conf.*, CS Press, Los Alamitos, Calif., 1993, pp. 49–55.

P.M. Athanas and A.L. Abbott, "Processing Images in Real Time on a Custom Computing Platform," in R.W. Hartenstein and M.Z. Servít, eds., *Field-Programmable Logic: Architectures, Synthesis, and Applications*, Springer-Verlag, Berlin, 1994, pp. 156–167.

P.M. Athanas and K.L. Pocek, eds., *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995.

P.M. Athanas and H.F. Silverman, "An Adaptive Hardware Machine Architecture for Dynamic Processor Reconfiguration," *Proc. Int'l Conf. Computer Design*, CS Press, Los Alamitos, Calif., 1991, pp. 397–400.

P.M. Athanas and H.F. Silverman, "Processor Reconfiguration through Instruction Set Metamorphosis: Architecture and Compiler," *Computer*, Vol. 26, No. 3, Mar. 1993, pp. 11–18.

J. Babb, R. Tessier, and A. Agarwal, "Virtual Wires: Overcoming Pin Limitations in FPGA-Based Logic Emulators," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 142–151.

S.L. Bade and B.L. Hutchings, "FPGA-Based Stochastic Neural Networks—Implementation," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 189–199.

J.P. Banâtre, D. Lavenier, and M. Vieillot, "From High Level Programming Model to FPGA Machines," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 119–125.

R.A. Bergamaschi, "High-Level Synthesis in a Production Environment: Methodology and Algorithms," in J.P. Mermet, ed., *Fundamentals and Standards in Hardware Description Languages*, Kluwer Academic Publishers, Boston, 1993, pp. 195–230.

N.W. Bergmann and J.C. Mudge, "Comparing the Performance of FPGA-Based Custom Computers with General-Purpose Computers for DSP Applications," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 164–172.

P. Bertin, *Mémoires Actives Programmables: Conception, Réalisation et Programmation*, PhD thesis, Université Paris 7, 1993.

P. Bertin, D. Roncin, and J. Vuillemin, "Programmable Active Memories: A Performance Assessment," in G. Borriello and C. Ebeling, eds., *Research on Integrated Systems: Proceedings of the '93 Symposium*, MIT Press, Cambridge, Mass., 1993, pp. 88-102.

P. Bertin and H. Touati, "PAM Programming Environments: Practice and Experience," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 133–139.

B. Box, "Field Programmable Gate Array Based Reconfigurable Preprocessor," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 40–49.

B. Box and J. Nieznanski, "Common Processor Element Packaging," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 39–46.

F. Brooks, *The Mythical Man-Month*, Addison-Wesley, Reading, Mass., 1975. (The notion of "second-system effect" seems to come from Brooks, although this precise definition comes from *The Hacker's Dictionary*, by Guy L. Steele, Jr.)

S.D. Brown et al., *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, Boston, 1992.

D.A. Buell and K.L. Pocek, eds., *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993.

D.A. Buell and K.L. Pocek, eds., *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994.

D.A. Buell and K.L. Pocek, "Custom Computing Machines: An Introduction," *J. of Supercomputing*, Vol. 9, 1995, pp. 219–230.

D.A. Buell and N. Shirazi, "A Splash 2 Tutorial," Tech. Report SRC-TR-92-087 (revised), SRC, Bowie, Md., 1993.

P.J. Burt and E.H. Adelson, "The Laplacian Pyramid as a Compact Image Code," *IEEE Trans. Comm.*, Vol. COM-31, No. 4, Apr. 1983, pp. 532–540.

R. Camposano et al., "The IBM High-Level Synthesis System," R. Camposano and Wayne Wolf, eds., *High Level Synthesis*, Kluwer Academic Publishers, Boston, 1991, pp. 79–104.

J.M. Carrera et al., "Architecture of a FPGA-Based Coprocessor: The PAR-1," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 20–29.

S. Casselman, "Virtual Computing and the Virtual Computer," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 43–49.

P.K. Chan and M.D.F. Schlag, "Architectural Tradeoffs in Field-Programmable-Device-Based Computing Systems," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 152–162.

L. Chen, "Fast Generation of Gaussian and Laplacian Image Pyramids Using an FPGA-based Custom Computing Platform," master's thesis, Virginia Polytechnic Inst., Blacksburg, Va., 1994.

H.A. Chow, H. Alnuweiri, and S. Casselman, "FPGA-Based Transformable Computers for Fast Digital Signal Processing," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 197–203.

Concurrent Logic Inc., *Cli6000 Series Field-Programmable Gate Arrays*, Concurrent Logic Inc., Sunnyvale, Calif., 1992.

N.G. Core et al., "Supercomputers and Biological Sequence Comparison Algorithms," *Computers and Biomedical Research*, Vol. 22, No. 6, 1989, pp. 497–515.

C.P. Cowen and S. Monaghan, "A Reconfigurable Monte-Carlo Clustering Processor (MCCP)," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 59–66.

C.E. Cox and W. Ekkehard Blanz, "Ganglion—a Fast Hardware Implementation of a Connectionist Classifier," IBM Research Report RJ8290, *Proc. 1991 IEEE Custom Integrated Circuits Conf.*, IEEE Press, Piscataway, N.J., 1991, pp. 6.5.1–6.5.4.

S.A. Cuccaro and C.F. Reese, "The CM-2X: A Hybrid CM-2/Xilinx Prototype," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 121–131.

M. Dahl et al., "Emulation of the Sparcle Microprocessor with the MIT Virtual Wires Emulation System," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 14–23.

M. Dao et al., "Acceleration of Template Based Ray Casting for Volume Visualization Using FPGAs," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 116–124.

J. Darnauer et al., "A Field Programmable Multi-Chip Module (FPMCM)," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 1–11.

Datacube, Inc., *The MaxVideo 200 Reference Manual*, Datacube, Inc., Danvers, Mass., 1994.

A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 31–40.

A.S. Deshpande, D.S. Richards, and W.R. Pearson, "A Platform for Biological Sequence Comparison on Parallel Computers," *CABIOS*, Vol. 7, No. 2, April 1991, p. 237.

P.M. Dew, R.A. Earnshaw, and T.R. Heywood, eds., *Parallel Processing for Computer Vision and Display*, Addison-Wesley, Reading, Mass., 1989.

P. Dhaussy et al., "Global Control Synthesis for an MIMD/FPGA Machine," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 72–82.

T. Drayer et al., "MORRPH: A MOdular and Reprogrammable Real-Time Processing Hardware," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 11–19.

R.O. Duda and P.E. Hart, "Use of the Hough Transform to Detect Lines and Curves in Pictures," *Comm. of the ACM*, Vol. 15, 1972, pp. 11–15.

J.G. Eldredge and B.L. Hutchings, "Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 180–189.

R. Elliott, "Hardware Implementation of a Straight Line Detector for Image Processing," master's thesis, Virginia Polytechnic Inst., Blacksburg,Va., 1993.

Federal Bureau of Investigation, *The Science of Fingerprints: Classification and Uses*, U.S. Govt. Printing Office, Washington, D.C., 1984.

P.W. Foulk, "Data Folding in SRAM Configurable FPGAs," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 163–172.

P.C. French and R.W. Taylor, "A Self-Reconfiguring Processor," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 50–60.

K.A. Frenkel, "The Human Genome Project and Informatics," *Comm. of the ACM*, Vol. 34, No. 11, 1991, pp. 41–51.

D. Gajski, ed., *Silicon Compilation*, Addison-Wesley, Reading, Mass., 1988.

D. Galloway, "The Transmogrifier C Hardware Description Language and Compiler for FPGAs," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 136–144.

G.J. Gent, S.R. Smith, and R.L. Haviland, "An FPGA-Based Custom Coprocessor for Automatic Image Segmentation Applications," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 172–180.

M. Gokhale, W. Holmes, and K. Iobst, "Processing in Memory: The Terasys Massively Parallel Processor Array," *Computer*, Vol. 28, No. 4, Apr. 1995, pp. 23–31.

M. Gokhale et al., "Building and Using a Highly Parallel Programmable Logic Array," *Computer*, Vol. 24, No. 1, Jan. 1991, pp. 81–89.

M. Gokhale et al., "The Logic Description Generator," Tech. Report SRC-TR-90-011, SRC, Bowie, Md., 1990.

M. Gokhale and R. Minnich, "FPGA Programming in a Data Parallel C," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 94–102.

M. Gokhale and B. Schott, "Data Parallel C on a Reconfigurable Logic Array," *J. of Supercomputing*, Vol. 9, 1995, pp. 291–314.

J.P. Gray and T.A. Kean, "Configurable Hardware: A New Paradigm for Computation," C.L. Seitz, ed., *Advanced Research in VLSI*, MIT Press, Cambridge, Mass., 1989, pp. 279–295.

H. Grünbacher and R.W. Hartenstein, eds., *Field Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, Springer-Verlag, Berlin, 1993. (Lecture Notes in Computer Science #705).

S.A. Guccione and M.J. Gonzalez, "A Data-Parallel Programming Model for Reconfigurable Architectures," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 79–88.

R.K. Gupta and C.N. Coelho Jr., "Program Implementation Schemes for Hardware-Software Systems," *Computer*, Vol. 27, No. 1, Jan. 1994, pp. 48–55.

J. Hadley and B. Hutchings, "Design Methodologies for Partially Reconfigured Systems," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 78–84.

R.W. Hartenstein, R. Kress, and H. Reinig, "A Reconfigurable Data-Driven ALU for Xputers," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 139–147.

S. Hauck and G. Borriello, "Pin Assignment for Multi-FPGA Systems," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 11–14.

K. Hayashi et al., "Reconfigurable Real-Time Signal Transport System Using Custom FPGAs," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 68–75.

B.U. Heeb, *Debora: A System for the Development of Field-Programmable Hardware, and Its Application to a Reconfigurable Computer*, PhD thesis, VDF, Informatik Dissertationen 45, ETH Zürich, Zürich, Switzerland, 1993.

B.U. Heeb and C. Pfister, "Chameleon, a Workstation of a Different Color," in H. Grünbacher and R.W. Hartenstein, eds., *Field Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, Springer-Verlag, Berlin, 1993, pp. 152–161.

J.H. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kauffmann Publishers, San Mateo, Calif., 1990.

H.-J. Herpel et al., "A Reconfigurable Computer for Embedded Control Applications," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 111–121.

Sir W.J. Herschel, *The Origin of Fingerprinting*, AMS Press, New York, 1974.

W.D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Mass., 1986.

D.T. Hoang, "A Systolic Array for the Sequence Alignment Problem," Tech. Report CS-92-22, Brown Univ., Providence, R.I., 1992.

D.T. Hoang, "Searching Genetic Databases on Splash 2," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 185–192.

D.T. Hoang and D.P. Lopresti, "FPGA Implementation of Systolic Sequence Alignment," in H. Grünbacher and R.W. Hartenstein, eds., *Field Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, Springer-Verlag, Berlin, 1993, pp. 183–191.

H. Högl et al., "Enable++: A Second Generation FPGA Processor," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 45–53.

P.V.C. Hough, "A Method and Means for Recognizing Complex Patterns," U.S. Patent No. 3,069,654, 1962.

R.P. Hughey, *Programmable Systolic Arrays*, PhD thesis CS-91-34, Brown Univ., Providence, R.I., 1991.

IEEE Standard VHDL Language Reference Manual, Std 1076-1987, IEEE Press, New York, 1988.

IEEE Standard VHDL Language Reference Manual, Std 1076-1992, IEEE Press, New York, 1992.

C. Iseli and E. Sanchez, "Spyder: A Reconfigurable VLIW Processor Using FPGAs," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 17–25.

C. Iseli and E. Sanchez, "A C++ Compiler for FPGA Custom Execution Units Synthesis," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 173–179.

C. Iseli and E. Sanchez, "Spyder: A SURE, SUperscalar and REconfigurable, Processor," *J. of Supercomputing*, Vol. 9, 1995, pp. 231–252.

B. Jahne, *Digital Image Processing*, Springer-Verlag, New York, 1991.

A.K. Jain, "Advances in Statistical Pattern Recognition," in *Pattern Recognition Theory and Applications*, P.A. Devijer and J. Kittler, eds., Springer-Verlag, New York, 1987, pp. 1–19.

A. Jantsch et al., "A Case Study on Hardware/Software Partitioning," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 111–119.

C. Jones et al., "Issues in Wireless Video Coding Using Run-Time-Reconfiguration FPGAs," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 85–89.

R. Jones, "Protein Sequence and Structure Comparison on Massively Parallel Computers, *Int'l J. of Supercomputer Applications*, Vol. 6, No. 2, 1992, pp. 138–146.

T.A. Kean, *Configurable Logic: A Dynamically Programmable Cellular Architecture and Its VLSI Implementation*, PhD thesis, Univ. of Edinburgh Dept. of Computer Science, Edinburgh, Scotland, UK, 1988.

T.A. Kean and C. Carruthers, "Bipolar CAL Chip Doubles the Speed of FPGAs," in W. Moore and W. Luk, eds., *FPGAs*, Abingdon EE & CS Books, Abingdon, England, UK, 1991, pp. 46–53.

T.A. Kean and J.P. Gray, "Configurable Hardware: Two Case Studies of Micrograin Computation," *J. of VLSI Signal Processing*, Vol. 2, 1990, pp. 9–16.

A. Kopser, "Splash 2: Architectural Motivation," tech. report, SRC, Bowie, Md., 1991.

H.T. Kung, "Why Systolic Architectures?" *Computer*, Vol. 15, 1982, pp. 37–46.

H.T. Kung and C.E. Leiserson, "Systolic Arrays for VLSI," in C.A. Mead and L.C. Conway, eds., *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980, pp. 271–292.

E.S. Lander, R. Langridge, and D.M. Saccocio, "Computing in Molecular Biology: Mapping and Interpreting Biological Information," *Computer*, Vol. 24, No. 11, Nov. 1991, pp. 6–13.

E.S. Lander, R. Langridge, and D.M. Saccocio, "Mapping and Interpreting Biological Information," *Comm. of the ACM*, Vol. 34, 1991, pp. 32–39.

H.C. Lee and R.E. Gaensslen, *Advances in Fingerprint Technology*, Elsevier, New York, 1991.

E. Lemoine and D. Merceron, "Run Time Reconfiguration of FPGA for Scanning Genomic Databases," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 90–98.

A. Lew and R. Halverson, "A FCCM for Dataflow (Spreadsheet) Programs," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 2–10.

D.M. Lewis, M.H. van Ierssel, and D.H. Wong, "A Field Programmable Accelerator for Compiled-Code Applications," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 60–68.

J. Li and C.K. Cheng, "Routability Improvement Using Dynamic Interconnect Architecture," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 61–67.

X.-P. Ling and H. Amano, "WASMII: A Data Driven Computer on a Virtual Hardware," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 33–43.

X.-P. Ling and H. Amano, "WASMII: A MPLD with Data Driven Control on a Virtual Hardware," *J. of Supercomputing*, Vol. 9, 1995, pp. 253–276.

R.J. Lipton and D.P. Lopresti, "A Systolic Array for Rapid String Comparison," *Proc. 1985 Chapel Hill Conf. VLSI*, Computer Science Press, Rockville, Md., 1985, pp. 363–376.

D.P. Lopresti, *Discounts for Dynamic Programming with Applications in VLSI Processor Arrays*, PhD thesis, Princeton Univ., Princeton, N.J., 1987.

D.P. Lopresti, "Fast Dictionary Searching on Splash," tech. report, SRC, Bowie, Md., 1991.

D.P. Lopresti, "Rapid Implementation of a Genetic Sequence Comparator Using Field Programmable Logic Arrays," in C.H. Séquin, ed., *Advanced Research in VLSI*, MIT Press, Cambridge, Mass., 1991, pp. 138–152.

D.P. Lopresti and R.J. Lipton, "Comparing Long Strings on a Short Systolic Array," Tech. Report CS-TR-026-86, Princeton Univ., Princeton, N.J., 1986.

M.E. Louie and M.D. Ercegovac, "A Digit-Recurrence Square Root Implementation for Field Programmable Gate Arrays," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 178–184.

M.E. Louie and M.D. Ercegovac, "A Variable Precision Square Root Implementation for Field Programmable Gate Arrays," *J. of Supercomputing*, Vol. 9, 1995, pp. 315–336.

W. Luk, "Pipelining and Transposing Heterogeneous Array Designs," *J. of VLSI Signal Processing*, Vol. 5, 1993, pp. 7–20.

W. Luk, "A Declarative Approach to Incremental Custom Computing," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 164–172.

W. Luk, V. Lok, and I. Page, "Hardware Acceleration of Divide-and-Conquer Paradigms: A Case Study," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 192–202.

W. Luk, T. Wu, and I. Page, "Hardware-Software Codesign of Multidimensional Programs," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 82–91.

P. Marchal and E. Sanchez, "CAFCA (Compact Accelerator for Cellular Automata): The Metamorphosable Machine," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 66–72.

W.J. Masek and M.S. Paterson, "How to Compute String-Edit Distances Quickly," in *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, D. Sankoff and J. Kruskal, eds., Addison-Wesley, Reading, Mass., 1983.

MasPar, Inc., *MasPar Application Language Reference Manual*, MasPar, Inc., Sunnyvale, Calif., 1990.

J.T. McHenry, "Dictionary Search Application on Splash," tech. report, SRC, Bowie, Md., 1991.

J.T. McHenry and A. Kopser, "Keyword Searching on Splash," tech. report, SRC, Bowie, Md., 1991.

Mead Data Central, *LEXIS Quick Reference*, Mead Data Central, New York, 1976.

R. Meier, "Rapid Prototyping of a RISC Architecture for Implementation in FPGAs," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 190–196.

P.J. Menchini, "An Introduction to VHDL," in J.P. Mermet, ed., *Fundamentals and Standards in Hardware Description Languages*, Kluwer Academic Publishers, Boston, 1993, pp. 359–384.

B. Miller, "Vital Signs of Identity," *IEEE Spectrum*, Vol. 31, No. 2, Feb. 1994, pp. 22–30.

G. Milne et al., "Realizing Massively Concurrent Systems on the SPACE Machine," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 26–33.

L. Moll, J. Vuillemin, and P. Boucard, "High-Energy Physics on DEC PeRLe-1 Programmable Active Memory," *Proc. FPGA95*, ACM, ACM Press, New York, 1995, pp. 47–52.

S. Monaghan and C.P. Cowen, "Reconfigurable Multi-Bit Processor for DSP Applications in Statistical Physics," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 103–111.

W. Moore and W. Luk, eds., *FPGAs*, Abingdon EE & CS Books, Abingdon, England, UK, 1992. (Proc., Oxford 1991 Int'l Workshop on Field Programmable Logic and Applications.)

W. Moore and W. Luk, eds., *J. of VLSI Signal Processing*, 1993, (Special Issue on Field-Programmable Gate Arrays.)

W. Moore and W. Luk, eds., *More FPGAs*, Abingdon EE & CS Books, Abingdon, England, UK, 1994. (Proc., Oxford 1993 Int'l Workshop on Field Programmable Logic and Applications.)

Q. Motiwala, "Optimizations for Acyclic Dataflow Graphs for Hardware-Software Codesign," master's thesis, Virginia Polytechnic Inst., Blacksburg, Va., 1994.

Nat'l Library of Medicine, *MEDLARS, The Computerized Literature Retrieval Services of the Nat'l Library of Medicine*, Publication NIH 79-1286, U.S. Dept. of Health, Education and Welfare, Washington, D.C., 1979.

M. Newman, W. Luk, and I. Page, "Constraint-Based Hierarchical Hardware Compilation of Parallel Programs," in R.W. Hartenstein and M.Z. Servít, eds., *Field-Programmable Logic: Architectures, Synthesis, and Applications*. Springer-Verlag, Berlin, 1994, pp. 220–229.

R.J. Offen, *VLSI Image Processing*, McGraw-Hill, New York, 1985.

J.K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, Mass., 1994.

I. Page and W. Luk, "Compiling Occam in FPGAs," in W. Moore and W. Luk, eds., *FPGAs*, Abingdon EE & CS Books, Abingdon, England, UK, 1991, pp. 271–283.

D.L. Perry, *VHDL*, McGraw-Hill, New York, 1991.

D.L. Perry, *VHDL*, McGraw-Hill, New York, 2nd ed., 1994.

W.K. Pratt, *Digital Image Processing*, Wiley, New York, 1978.

D.V. Pryor, M.R. Thistle, and N. Shirazi, "Text Searching on Splash 2," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, 172–178.

G. Purcell and D. Mar, "SCOUT: Information Retrieval from Full-Text Medical Literature," Knowledge Systems Lab. Report KSL-92-35, Stanford Univ., Palo Alto, Calif., 1992.

G.M. Quénot et al., "A Reconfigurable Compute Engine for Real-Time Vision Automata Prototyping," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 91–101.

R. Rachakonda, "Region Detection and Labeling in Real-time Using a Custom Computing Platform," master's thesis, Virginia Polytechnic Inst., Blacksburg, Va., 1994.

F. Raimbault et al., "Fine Grain Parallelism on a MIMD Machine Using FPGAs," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 2–9.

N.K. Ratha, A.K. Jain, and D.T. Rover, "Fingerprint Matching on Splash 2," tech. report, Dept. of Computer Science, Michigan State Univ., East Lansing, Mich., Mar. 1994.

N.K. Ratha, A.K. Jain, and D.T. Rover, "Convolution on Splash 2," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 204–213.

A. Rosenfeld and A. Kak, *Digital Picture Processing*, 2nd ed., Academic Press, New York, 1982.

G. Salton, *Automatic Text Processing*, Addison-Wesley, Reading, Mass., 1989.

G. Salton and M.J. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York, 1983.

D. Sankoff and J. Kruskal, eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, Mass., 1983.

J. Schlesinger and M. Gokhale, dBC Reference Manual. Tech. Report SRC-TR-92-068, Revision 2, SRC, Bowie, Md., 1993.

H. Schmit et al., "Behavioral Synthesis for FPGA-Based Computing," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 125–133.

H. Schmit and D. Thomas, "Implementing Hidden Markov Modelling and Fuzzy Controllers in FPGAs," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 214–221.

J. Serra, *Image Analysis and Mathematical Morphology*, Academic Press, London, 1982.

M. Shand, "Flexible Image Acquisition Using Reconfigurable Hardware," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 125–134.

M. Shand, P. Bertin, and J. Vuillemin, "Hardware Speedups for Long Integer Multiplication," *ACM Symp. Parallel Algorithms and Architectures*, ACM, ACM Press, New York, 1990, pp. 138–145.

N. Shirazi, "Implementation of a 2-D Fast Fourier Transform on an FPGA-based Computing Platform," master's thesis, Virginia Polytechnic Inst., 1995.

N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating-Point Arithmetic on FPGA-based Custom Computing Machines," *Proc. IEEE Symp. FPGAs for Custom Computing*, CS Press, Los Alamitos, Calif., Apr. 1995, pp. 155–162.

S. Singh, "Architectural Description for FPGA Circuits," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 145–154.

S. Singh and P. Bellec, "Virtual Hardware for Graphics Applications Using FPGAs," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 49–59.

N. Sitkoff et al., "Implementing a Genetic Algorithm on a Parallel Custom Computing Machine," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 180–187.

C. Stanfill and B. Kahle, "Parallel Free-Text Search on the Connection Machine System," *Comm. of the ACM*, Vol. 29, No. 12, 1986, pp. 1229–1239.

J. Stigliani, *Writing SBus Device Drivers*, Sun Microsystems, Inc., Mountain View, Calif., 1990.

Synopsys, Inc., *Design Compiler Reference Manual*, Synopsys, Inc., Mountain View, Calif., 1991.

Synopsys, Inc., *VHDL Compiler Reference Manual*, Synopsys, Inc., Mountain View, Calif., 1991.

Synopsys, Inc., *FPGA Compiler Reference Manual*, Synopsys, Inc., Mountain View, Calif., 1994.

A. Tarmaster, "Median and Morphological Filtering of Images in Real Time Using an FPGA-based Custom Computing Platform," master's thesis, Virginia Polytechnic Inst., Blacksburg, Va., 1994.

Texas Instruments Inc., *The SN74ACT8800 Family Data Manual (SCSS006A)*, Texas Instruments Inc., Dallas, 1988.

Thinking Machines, Inc., *C* Programming Guide*, Thinking Machines, Inc., Cambridge, Mass., 1993.

D.E. Thomas and P.R. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, Boston, 1991.

S.M. Trimberger, ed., *Field Programmable Gate Array Technology*, Kluwer Academic Publishers, Boston, 1994.

Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983, U.S. Department of Defense, Washington, D.C., Feb. 1983.

L. Uhr, ed., *Parallel Computer Vision*, Academic Press, New York, 1987.

M. van Daalen, P. Jeavons, and J. Shawe-Taylor, "A Stochastic Neural Architecture That Exploits Dynamically Reconfigurable FPGAs," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 202–212.

D.E. Van den Bout, "The Anyboard: Programming and Enhancements," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 68–78.

G. VanDerWal and P. Burt, "A VLSI Pyramid Chip for Multiresolution Image Analysis," *Int'l J. of Computer Vision*, Vol. 8, No. 3, 1992, pp. 177–189.

R. Vogt, *Automatic Generation of Morphological Set Recognition Algorithms*, Springer-Verlag, New York, 1989.

J. Vuillemin et al., "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Trans. VLSI Systems*, to be published in Mar. 1996.

M. Wazlowski et al., "PRISM II: Compiler and Architecture," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 9–17.

J.H. Wegstein, *An Automated Fingerprint Identification System*, Special Publication 500–89, Nat'l Bureau of Standards, Washington, D.C., 1982.

R. Wieler, Z. Zhang, and R. McLeod, "Emulating Static Faults Using a Xilinx Based Emulator," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, California, 1995, pp. 110–115.

M. Wirthlin and B. Hutchings, "A Dynamic Instruction Set Computer," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 99–107.

M.J. Wirthlin, B.L. Hutchings, and K.L. Gilson, "The Nano Processor: A Low Resource Reconfigurable Processor," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 23–31.

D. Wo and K. Forward, "Compiling to the Gate Level for a Reconfigurable Co-Processor," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1994, pp. 147–155.

L.F. Wood, "High Performance Analysis and Control of Complex Systems Using Dynamically Reconfigurable Silicon and Optical Fiber Memory," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1993, pp. 132–142.

Xilinx, Inc., *The Programmable Gate Array Data Book*, Xilinx, Inc., San Jose, Calif., 1993.

Xilinx, Inc., *The XC4000 Data Book*, Xilinx, Inc., San Jose, Calif. 1994.

C.-C. Yeh, C.-H. Wu, and J.-Y. Juang, "Design and Implementation of a Multicomputer Interconnection Using FPGAs," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, CS Press, Los Alamitos, Calif., 1995, pp. 56–60.

Index

Contributors

A. Lynn Abbott, Bradley Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061. 703-231-4472
Jeffrey M. Arnold, Center for Computing Sciences, 17100 Science Drive, Bowie, Maryland 20715. 301-805-7479
Peter Athanas, Bradley Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061. 703-231-7010
Duncan A. Buell, Center for Computing Sciences, 17100 Science Drive, Bowie, Maryland 20715. 301-805-7372
Maya Gokhale, David Sarnoff Research Center, CN 5300, Princeton, New Jersey 08543. 609-734-3119
Dzung T. Hoang, Department of Computer Science, Duke University, Durham, North Carolina 27706. 919-660-6598
Anil Jain, Department of Computer Science, Michigan State University, East Lansing, Michigan 48824. 517-353-5150
Walter J. Kleinfelder, Center for Computing Sciences, 17100 Science Drive, Bowie, Maryland 20715. 301-805-7355
Daniel V. Pryor, Center for Computing Sciences, 17100 Science Drive, Bowie, Maryland 20715. 301-805-7407
Nalini Ratha, Department of Computer Science, Michigan State University, East Lansing, Michigan 48824. c/o A. Jain 517-353-5150
Diane Rover, Department of Electrical Engineering, Michigan State University, East Lansing, Michigan 48824. 517-353-7735
Nabeel Shirazi, Bradley Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061. c/o P. Athanas 703-231-7010
Mark R. Thistle, Center for Computing Sciences, 17100 Science Drive, Bowie, Maryland 20715. 301-805-7413

IEEE Computer Society Press Editorial Board

Advances in Computer Science and Engineering

The IEEE Computer Society Press Advances Board seeks manuscripts that describe new and significant advances in computer science and engineering. Although immediate application is not necessary, ultimate application to advanced computing systems is an important quality. Publications represent technically substantive and clear expositions of innovative ideas.

Additional Advances Board Titles

A Probabilistic Analysis of Test-Response Compaction
Slawomir Pilarski and Tiko Kameda

The Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions
Igor Tartalja and Veljko Milutinović

The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions
Igor Tartalja and Veljko Milutinović

Advanced Multimicroprocessor Bus Architectures
Janusz Zalewski

IEEE COMPUTER SOCIETY
50 YEARS OF SERVICE • 1946-1996

http://www.computer.org

Press Activities Board

IEEE Computer Society Press Publications

The world-renowned Computer Society Press publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available in two formats: 100 percent original material by authors preeminent in their field who focus on relevant topics and cutting-edge research, and reprint collections consisting of carefully selected groups of previously published papers with accompanying original introductory and explanatory text.

Submission of proposals: For guidelines and information on CS Press books, send e-mail to csbooks@computer.org or write to the Acquisitions Editor, IEEE Computer Society Press, P.O. Box 3014, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314. Telephone +1 714-821-8380. FAX +1 714-761-1784.

IEEE Computer Society Press Proceedings

The Computer Society Press also produces and actively promotes the proceedings of more than 130 acclaimed international conferences each year in multimedia formats that include hard and softcover books, CD-ROMs, videos, and on-line publications.

For information on CS Press proceedings, send e-mail to csbooks@computer.org or write to Proceedings, IEEE Computer Society Press, P.O. Box 3014, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314. Telephone +1 714-821-8380. FAX +1 714-761-1784.

Additional information regarding the Computer Society, conferences and proceedings, CD-ROMs, videos, and books can also be accessed from our web site at www.computer.org.

3/21/96

Splash 2

FPGAs in a Custom Computing Machine

edited by Duncan A. Buell, Jeffrey M. Arnold, and Walte

Details the complete Splash 2 project–the hardware and software systems, their architecture and implementation, and the design process by which the architecture evolved from an earlier version machine. In addition to the description of the machine, this book explains why Splash 2 was engineered. It illustrates several applications in detail, allowing you to gain an understanding of the capabilities and the limitations of this kind of computing device.

The Splash 2 program is significant for two reasons. First, it is part of a complete computer system that achieves supercomputer like performance on a number of different applications. The second significant aspect is that this large system is capable of performing real computations on real problems. In order to understand what happens when the application programmer designs the processor architecture of the machine that executes his programs, it is necessary to see the system as a whole. This book looks in-depth at one of the handful of data points in the design space of this new kind of machine.

Contents:

Published by the IEEE Computer Society Press
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314

ISBN 0-8186-7413-X
90000>
9 780818 674136

IEEE Computer Society Press Order Number BP07413
Library of Congress Number 95-47397
ISBN 0-8186-7413-X