# Efficient Fingerprint-based User Authentication for Embedded Systems

Pallav Gupta[†], Srivaths Ravi[‡], Anand Raghunathan[‡], Niraj K. Jha[†]
[†] Dept. of Electrical Engineering, Princeton University, Princeton, NJ 08544, USA
[‡] NEC Laboratories America Inc., Princeton, NJ 08540, USA

## ABSTRACT

User authentication, which refers to the process of verifying the identity of a user, is becoming an important security requirement in various embedded systems. While conventional solutions for user authentication have relied on password-based mechanisms, they are increasingly being replaced by biometric technologies such as fingerprint, face, and voice recognition, which are known to provide higher levels of security for user authentication. This paper investigates the problem of supporting efficient fingerprint-based user authentication in embedded systems. For improving the performance of fingerprint-based authentication, we propose hardware/software enhancements that include a generic set of custom instruction extensions to an embedded processor's instruction set architecture, a memory-aware software re-design, and fixed-point arithmetic. We believe that the custom instruction set extensions proposed in this work are generic enough to speed up many fingerprint matching algorithms and even other biometric algorithms. Our experiments with an open-source, high-fidelity fingerprint authentication algorithm and a testbed featuring a commercial extensible processor show that performance is improved by a factor of 10.4X when using the proposed enhancements, while incurring modest overheads.

**Categories and Subject Descriptors:** J.6[Computer-Aided Engineering]: Computer Aided Design

**General Terms:** Design, Embedded Systems, Security.

**Keywords:** User authentication, fingerprint, extensible processors.

## 1. INTRODUCTION

Embedded electronic systems are ubiquitously used to capture, store, manipulate, and access data of a sensitive nature (*e.g.*, cell phones, personal digital assistants (PDAs), smart cards). The growing number of instances of breaches in information security in the last few years has created a compelling case for further efforts towards secure embedded systems. Recent advances in embedded system security have addressed some issues including secure communication, secure information storage, and tamper resistance [1, 2]. However, user-to-device authentication typically forms the weakest security link in the total security chain of most embedded systems, and is often poorly addressed.

Currently, most solutions to user authentication use surrogate representations of a person's identity, such as passwords and personal identification numbers and token cards. These approaches suffer from several drawbacks, including insufficient security and inconvenience to users [3]. *Biometrics*, which refer to the automatic recognition of people based on their distinctive physiological (*e.g.*, face, fingerprint, iris) and behavioral (*e.g.*, signature, gait) characteristics, could form a component of effective user identification solutions, because they intrinsically and reliably represent the individual's bodily identity [4]. Biometric characteristics cannot be lost or forgotten; they are extremely difficult to copy, share, and distribute; they require the person being authenticated to be physically present at the time and point of authentication. Of these, fingerprint-based biometric authentication is increasingly being accepted as a viable choice for user authentication in many systems such as personal laptops, enterprise systems, *etc.*, due to their high accuracy (lower false accept and reject rates relative to voice and face-based solutions) and lower cost (relative to techniques such as iris recognition) [4]. However, high-fidelity fingerprint-based authentication can be very compute-intensive, since various signal processing techniques need to be used to compensate for noisy or inaccurate measurements (due to cost constraints, embedded systems such as mobile phones can afford to employ only low-to-moderate resolution sensors). Furthermore, processing times for biometric authentication must be in the order of a few seconds. Because such systems are severely resource-constrained, performance of software implementations of high-fidelity algorithms can be extremely poor.

In this paper, we enhance the architecture of an embedded system to address this issue. The main contribution of this work is that we provide a set of hardware/software enhancements to speed up user authentication. The proposed techniques push the efficiency envelope of high-fidelity fingerprint-based authentication, and can be applied to a wide range of fingerprint, and even other biometric, algorithms. To the best of our knowledge, this is the first investigation of the use of custom instructions to accelerate biometric authentication.

The rest of this paper is organized as follows. Section 2 discusses the steps involved in fingerprint-based user authentication and presents an example of such a system, while Section 3 surveys related work. Section 4 details the hardware/software enhancements that are used to speed up user authentication. Section 5 concludes this paper.

## 2. PRELIMINARIES

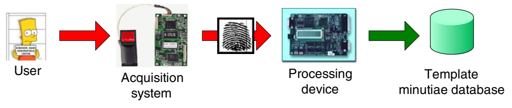
In this section, we describe the steps involved in fingerprint-based authentication and an example implementation of such a system.

### 2.1 Fingerprint-based Authentication

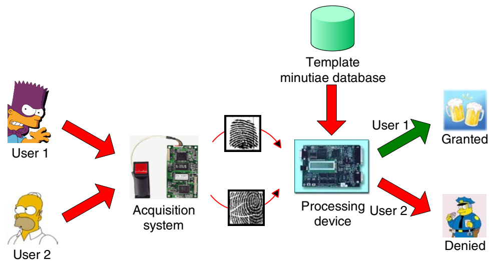
There are two types of fingerprint-based authentication techniques: graph-based and minutiae-based [4]. In this work, we concentrate on the latter because minutiae[1] are widely believed to be the most discriminating and reliable features of a fingerprint. In addition, the amount of information needed to be stored in the template database for fingerprint matching is smaller and the processing time is shorter than that of graph-based algorithms. This is important in embedded systems where processing resources are limited and saving energy is a major goal.

A fingerprint-based authentication system consists of two main steps: user enrollment and user authentication, which

[1]The term *minutiae* refers to features in a fingerprint such as ridge endings and bifurcations.



(a) User enrollment.



(b) User authentication.

**Figure 1: The two main components of a typical fingerprint-based authentication system.**

are shown in Figures 1(a) and 1(b), respectively. In the first step, an acquisition system captures an image of the user's fingerprint. A series of image processing procedures are then applied to the image to detect and extract the minutiae. The extracted minutiae are stored in a template database and the user is now considered enrolled. During user authentication, the user supplies a fingerprint image which is again processed to detect and extract the minutiae. These minutiae are then compared against the reference minutiae stored in the template database. A reference score is calculated based upon the number of minutiae that match. The user is considered authenticated if the score exceeds a specified threshold.

### 2.2 An Example System

In this work, we used a popular, open-source software implementation of the algorithm described in [5]. It is called Fingerprint Verification System (FVS) [6]. On top of this minutiae extraction algorithm, we implemented the local structure based minutiae matching algorithm described in [7].

The overall block diagram of FVS is shown in Figure 2. The first step in minutiae extraction involves normalizing the captured image to have a pre-specified mean and variance. Then, an orientation image is created which contains the coordinates of the ridges and furrows in the fingerprint. Next, a frequency image along the ridges is obtained by examining each pixel and those in its local neighborhood. The rationale is that, in a local neighborhood in which no minutiae are present, the gray levels between the ridges and furrows can be modeled as a sinusoidal wave. If this is not the case, then a minutia must be present. Based on the frequency image, a region mask is created where parts of the original input image are marked as recoverable or unrecoverable. Gabor filtering is performed on the recoverable parts to remove noise and highlight the ridges and furrows. Binarization is done on the pixels of the image to mark them as either black or white. Finally, the minutiae are extracted by examining each pixel and its immediate neighbors.

Once the minutiae have been extracted, they are compared against those in the template minutiae database to determine a match. In the local structure algorithm [7], one minutia each is chosen from the input and template minutiae databases, respectively. Their neighbors are compared to determine if the candidates match. If a specified number of neighbors match, the two minutiae are considered a match. The process is repeated for every pair of minutiae in the databases. That is, given input and template databases with $M$ and $N$ minutiae, respectively, $MN$ comparisons are required. Finally, a match score is calculated and if it exceeds a given threshold, a match is declared.

## 3. RELATED WORK

In this section, we survey various advances in fingerprint-based authentication. The relevant work can be broadly classified into two main areas. The first deals with the development of robust algorithms for minutiae detection, extraction, and matching. The second is centered around efficiency issues that arise when such an algorithm is deployed in a given system.

At the algorithm level, many solutions for fingerprint matching have been proposed that attempt to improve the overall authentication accuracy (lower false accepts and rejects). For example, works, such as those presented in [5], incorporate various filtering and compensation techniques to account for pose transformations and other deformations that may be present in a fingerprint. A rich survey of various fingerprint matching algorithms and their comparative analysis can be found in [8].

With security in resource-constrained embedded systems becoming important, researchers have attempted to integrate authentication technologies into embedded systems. In that context, a recent work [9] empirically shows that the processing workload of a software implementation of a gray-scale minutiae detection algorithm can be very significant while running on a PDA such as an iPAQ. Since a significant burden is due to the emulation of floating-point computations on the StrongARM processor that has no floating-point unit (FPU), the paper showed that the use of fixed-point arithmetic in core computations can provide considerable speedup. Based on the target embedded system and the fingerprint-based authentication algorithm used, other works have suggested hardware and software enhancements for acceleration. For example, the fingerprint verification module implemented in the embedded device ThumbPod [10] uses multiply-accumulate hardware to accelerate the Discrete Fourier Transform (DFT) computation used in minutiae detection and software optimizations that can reduce the number of DFT calculations performed in minutiae extraction.

## 4. ARCHITECTURAL ENHANCEMENTS

In this section, we present architectural enhancements that were used to speed up the performance of FVS running on an embedded processor (the T1050.3 Xtensa processor from Tensilica [11]). The processor was configured with separate two-way, 8KB instruction and data caches and a 32-bit multiplier. It was configured without an FPU because the addition of an FPU incurs a 60% area overhead compared to the base processor. This overhead is very significant for an embedded system. Figure 3 shows the experimental testbed used in our work. It includes three main components: (a) Xtensa emulation platform running at 33 MHz, (b) Sharp Zaurus PDA, and (c) AES3400 fingerprint sensor from Authentec [12] to capture fingerprint images. The fingerprint sensor is connected to the PDA, and the captured image is transferred to the emulation platform for processing by the FVS-based authentication software. Once processing is completed, the matching results are displayed on the PDA.

### 4.1 Performance Analysis

We profiled FVS on Xtensa's instruction set simulator (ISS). The test data was a set of fingerprint images ($256 \times 256$ pixels) of various people acquired from the fingerprint sensor. The ISS did not terminate after running for a week. This was because FVS is a highly floating-point compute-intensive application and the ISS was using software macros to emulate floating-point operations. Given the prohibitive time cost of running an ISS, we decided to run FVS directly on the emulation board to determine baseline performance. To address the above issue, the floating-point operations in FVS were converted to fixed-point operations. Fixed-point arithmetic utilizes the 32-bit integer arithmetic logic unit to perform the same computation at the expense of accuracy and the range of numbers that can be represented with a given precision. After carefully analyzing the code and the range of values that were produced for various fingerprint images, we decided to use two base formats, namely Q.22 and Q.28. In the Q.22 format, the lower 22 bits of a 32-bit integer are used to represent the fraction. The upper 10 bits are used to represent a signed integer. Thus, values between -512 and 511 can be represented with a precision of five to six decimal digits. Similarly, values

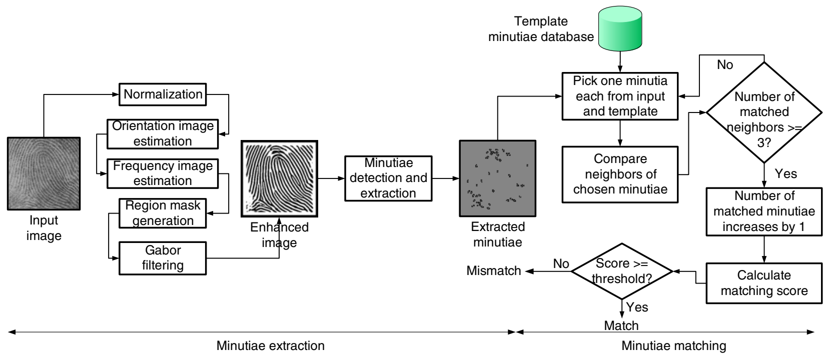

Template minutiae database
Normalization
Orientation image estimation
Frequency image estimation
Region mask generation
Gabor filtering
Input image
Enhanced image
Minutiae detection and extraction
Extracted minutiae
Pick one minutia each from input and template
Compare neighbors of chosen minutiae
No
Number of matched neighbors >= 3?
Yes
Number of matched minutiae increases by 1
Calculate matching score
Score >= threshold?
No
Mismatch
Yes
Match
Minutiae extraction
Minutiae matching

**Figure 2: The steps involved in minutiae detection, extraction, and matching.**

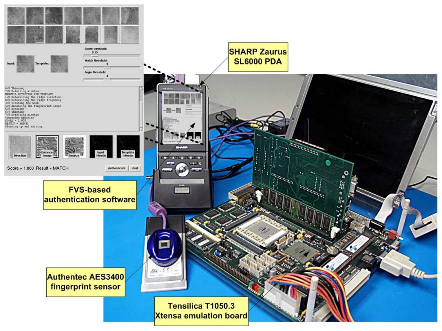

SHARP Zaurus SL6000 PDA
FVS-based authentication software
Authentec AES3400 fingerprint sensor
Tensilica T1050.3 Xtensa emulation board

**Figure 3: Experimental testbed for fingerprint-based authentication.**

between -8 and 7 can be represented with the Q.28 format. Note that overflow/underflow is possible although we did not observe any instance in our test suite of fingerprint images. In the event that it does occur, the errors will be local. If enough such errors do occur in a local neighborhood, then we may not be able to detect a potential minutia. However, not being able to detect one minutia among, say, fifty minutiae (typical for a sample fingerprint) does not pose a significant problem.

The basic operations of addition, subtraction, and multiplication were straightforward and easy to implement. Precision problems arose when trying to do fixed-point trigonometric computations such as *sine* and *cosine*. The popular table-lookup method proved to be insufficient in terms of accuracy. Higher accuracy could have been obtained with larger tables at the expense of a higher memory storage requirement. Instead, we used the CORDIC method [13], which is an iterative algorithm that calculates trigonometric values using addition and multiplication. However, the multiplicand in the multiplication is limited to powers of two. This translates to a shift operation and makes CORDIC very amenable to hardware implementation. All trigonometric functions and other mathematical functions (such as *exponential*, *square root*, *etc.*) were implemented using CORDIC.

After FVS was converted from floating-point to fixed-point, we tried to profile it on the ISS again. This time the simulation did complete and Figure 4 shows the flat execution profile. The functions, whose name begins with *Finger*, correspond to the various processing steps shown in Figure 2. As can be seen from the profile, 88% of the computation time is spent in the *cordit1* function which is called by *sincosCordic* to calculate the sine and cosine of a value. The *sincosCordic* function is utilized heavily in majority of the enhancement steps of Figure 2.

## 4.2 Instruction Set Extensions

Table 1 provides a list of instructions that were added to the base processor, their opcodes, their inputs and outputs, and a brief description of what they do. For example, the *cordit1* function was implemented as a custom instruction because it consumes 88% of the total execution time. The semantics of *cordit2* and *cordit3* are very similar to that of *cordit1*. Con-

| % time | cumulative seconds | calls | name |
|---|---|---|---|
| 88.15 | 91.12 | 50236516 | cordit1 |
| 3.07 | 94.28 | 25002546 | FingerEnhanceGabor |
| 3.06 | 97.45 | 25002546 | cordit2 |
| 2.27 | 99.79 | 2 | FingerGetFrequency |
| 1.55 | 101.40 | 2 | FingerDirectionLowPass |
| 0.57 | 101.99 | 50236516 | sincosCordic |
| 0.43 | 102.90 | 234058 | cordit3 |
| 0.30 | 103.21 | 2 | FingerOrientation |

**Figure 4: Profile of FVS using fixed-point operations.**

**Table 1: Custom Instructions for FVS**

| Name | Opcode | Inputs/Outputs | Description |
|---|---|---|---|
| cordit1 | $4'b0000$ | $r1$, $r2$, $r3/r1$, $r2$ | Take input in $r3$. Put $sin(r3)$ in $r2$. Put $cos(r3)$ in $r1$. |
| cordit2 | $4'b0001$ | $r1$, $r2$, $r3/r2$ | Take input in $r3$. Put $exp(r3)$ in $r2$. Internal state in $r1$. |
| cordit3 | $4'b0010$ | $r1$, $r2$, $r3/r2$ | Take input in $r3$. Put $atan(r3)$ in $r2$. Internal state in $r1$. |

sequently, they too were chosen to be implemented as custom instructions because virtually the same hardware can be used for their implementation.

A snippet of the FVS code, which utilizes the *cordit1* function, is shown in Figure 5(a). The pseudo-code of the *cordit1* function is shown in Figure 5(b). Figure 5(c) shows the register-transfer level (RTL) implementation that is functionally equivalent to the body of the $for$ loop highlighted in Figure 5(b). This implementation was translated to the Tensilica Instruction Extension (TIE) language. Figure 5(d) shows how the original code in Figure 5(a) was re-instrumented to utilize the *cordit1* custom instruction. The input registers are initialized using the $WT\#()$ macros. The body of the loop is replaced by a call to the custom instruction. Once the computation is complete, the $RT\#()$ macros are used to read the values from the registers and transfer their contents to the "original" program variables.

The TIE specification for the custom instructions was compiled by the TIE compiler from Tensilica and the RTL implementation for the custom processor was generated. The RTL implementation was then synthesized using Design Compiler from Synopsys to get an accurate measure of gate count. The total gate count for the custom instructions was 10,983. This represents approximately 10% area overhead when compared to the area of the embedded processor without custom instructions. The overhead is also much more modest than that of an FPU (60%). Using design tools from Altera, the new processor was downloaded onto the emulation board and the performance was re-evaluated.

## 4.3 Addressing Memory-related Issues

Fingerprint-based authentication is a data-dominated application. If we look back at Figure 2, we see that operations are being performed sequentially over the pixels of an image.

```
void sincosCordic(fp28 a, fp28* cosp, fp28 *sinp)
{
  ...
  *sinp = 0;
  *cosp = invGain1;
  cordit1(cosp, sinp, &a);
}
```

(a) Original code utilizing the *cordit1* function.

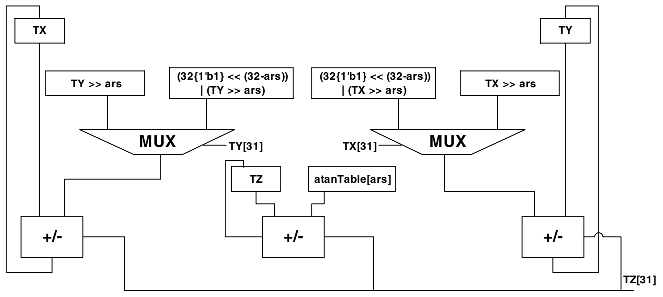
```
static void cordit1(fp28* x0, fp28* y0, fp28* z0)
{
  fp28 t, x, x1, y, z;
  int i;

  t = FP28_ONE; // t = 1
  x = *x0; y = *y0; z = *z0;

  for (i = 0; i < 32; ++i) {
   if (z >= 0) {
    x1 = FPSUB(x, FPMUL28(y,t)); // x1 = x - y*t
    y = FPADD(y, FPMUL28(x,t));  // y = y + x*t
    z = FPSUB(z, atanTable[i]);  // z = z + atanTable[i]
   }
   else {
    x1 = FPADD(x, FPMUL28(y,t)); // x1 = x + y*t
    y = FPSUB(y, FPMUL28(x,t));  // y = y - x*t
    z = FPADD(z, atanTable[i]);  // z = z + atanTable[i]
   }
   x = x1;
   t >>= 1; // shift right one bit
  }

  *x0 = x;
  *y0 = y;
  *z0 = z;
}
```

(b) Pseudo-code for the *cordit1* function.



TX
TY
TY >> ars
(32{1'b1} << (32-ars)) | (TY >> ars)
(32{1'b1} << (32-ars)) | (TX >> ars)
TX >> ars
MUX
TY[31]
TX[31]
MUX
TZ
atanTable[ars]
+/-
+/-
+/-
TZ[31]

(c) RTL specification for the *cordit1* custom instruction.

```
static void cordit1(fp28* x0, fp28* y0, fp28* z0)
{
  fp28 t, ta, tb, x, y, z;
  int i;
  x = *x0; y = *y0; z = *z0;

  // write registers
  WTX(x);
  WTY(y);
  WTZ(z);

  for (i = 0; i < 32; i++)
   cordit1(i);  // custom instruction call

  // read registers
  *x0 =  RTX();
  *y0 =  RTY();
  *z0 =  RTZ();
}
```

(d) Original code re-instrumented to utilize the *cordit1* custom instruction.

**Figure 5: Example to demonstrate custom instruction implementation and usage.**

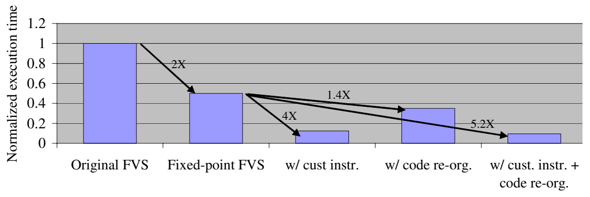


**Figure 6: Normalized execution times of FVS in five different scenarios.**

Because data cache and fingerprint image sizes were 8KB and 64KB, respectively, we observed significant memory traffic in moving the image data between memory and cache due to the sequential nature of the FVS algorithm. For example, the frequency image is calculated only after the orientation image for the entire image has been obtained. Clearly, this is a bottleneck.

In the FVS algorithm, computation is very localized. That is, to obtain a particular piece of information (orientation, frequency, *etc.*) about a pixel, it is only necessary to analyze the values of the pixels that are within a certain distance in a local neighborhood. This holds true for most of the computations in Figure 2. We can improve the cache data access patterns of the algorithm by re-organizing the code. The is done by dividing the image into blocks of 4KB. An image block of this size is then fetched from memory to cache. Then, the orientation and frequency images and the region mask are calculated and Gabor filtering is done on the block before proceeding to the next block.

### 4.4 Experimental Results

Figure 6 compares the normalized execution time of FVS in five different scenarios. The first and second bars show the performance of FVS with floating-point and fixed-point operations, respectively. In this case, a factor of 2X improvement was obtained. After the introduction of custom instructions and re-organization of the algorithm to reduce memory traffic, a further speedup of 5.2X was achieved. This amounts to roughly 10.4X speedup when using our proposed architectural enhancements. The individual speedups of using only custom instructions or code re-organization are also shown. The reason we do not see a significant speedup when using code re-organization only is that there is still memory traffic when dealing with pixels lying near the borders of the fingerprint images or the borders of the 4KB memory blocks.

## 5. CONCLUSIONS

In this paper, we presented an efficient architecture for performing fingerprint-based user authentication in embedded systems. For a popular, high-fidelity fingerprint-based authentication algorithm, we first developed various architectural enhancements that included fixed-point conversions, a low- overhead set of custom instructions, and memory-aware code re-organization to significantly boost overall performance. We believe that the biometric instruction extensions proposed in this work are generic enough to be utilized in other fingerprint algorithms and even other biometric algorithms.

## 6. REFERENCES

[1] R. York, *A New Foundation for CPU Systems Security.* ARM Limited, 2003.

[2] P. Kocher, R. B. Lee, G. McGraw, A. Raghunathan, and S. Ravi, “Security as a new dimension in embedded system design,” in *Proc. Design Automation Conf.*, June 2004, pp. 753–760.

[3] I. Armstrong, “Passwords exposed: Users are the weakest link,” in *http://www.scmagazine.com*, June 2003.

[4] A. Jain, R. Bolle, and S. Pankanti, Eds., *Biometrics: Personal Identification in Networked Society.* Boston, MA: Kluwer Academic, 2002.

[5] L. Hong, Y. Wan, and A. K. Jain, “Fingerprint image enhancement: Algorithm and performance evaluation,” *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 20, no. 8, pp. 777–789, Aug. 1998.

[6] “FVS website.” http://fvs.sourceforge.net

[7] S. Yang and I. M. Verbauwhede, “A secure fingerprint matching technique,” in *Proc. Wkshp. Biometrics Applications & Methods*, Nov. 2003, pp. 89–94.

[8] D. Maltoni, D. Maio, A. K. Jain, and S. Prabhakar, *Handbook of Fingerprint Recognition.* New York, NY: Springer, 2003.

[9] T. Y. Tang, Y. S. Moon, and K. C. Chan, “Efficient implementation of fingerprint verification for embedded systems using fixed-point arithmetic,” in *Proc. Symp. Applied Computing*, Mar. 2004, pp. 821–825.

[10] P. Schaumont and I. Verbauwhede, “Thumbpod puts security under your thumb,” *Xilinx Xcell J.*, 2003.

[11] *Xtensa Application Specific Microprocessor Solutions - Overview Handbook.* Tensilica Inc., 2001. http://www.tensilica.com

[12] “Authentec website.” http://www.authentec.com

[13] R. Andraka, “A survey of CORDIC algorithms for FPGAs,” in *Proc. Int. Symp. Field-Programmable Gate Arrays*, Feb. 1998, pp. 191–200.