Microcoded Coprocessor for Embedded Secure Biometric Authentication Systems

Shenglin Yang
UCLA Dept of EE
Los Angeles, CA 90095
+1-310-267-4940
shengliny@ee.ucla.edu

Patrick Schaumont
UCLA Dept of EE
Los Angeles, CA 90095
+1-310-267-4940
schaum@ee.ucla.edu

Ingrid Verbauwhede
UCLA Dept of EE
Los Angeles, CA 90095
+1-310-794-5209
ingrid@ee.ucla.edu

ABSTRACT
We design and implement a cryptographic biometric authentication system using a microcoded architecture. The secure properties of the biometric matching process are obtained by means of a fuzzy vault scheme. The algorithm is implemented in a reprogrammable, microcoded coprocessor called FV16. We present the micro-architecture of FV16 as well as a dedicated assembler for this architecture. Our coprocessor can be attached to an ARM processor, and offers a 83-fold cycle count improvement when the fuzzy vault algorithm is migrated from embedded ARM software (13.8 million cycles) to the FV16 coprocessor (166 thousand cycles).

Categories and Subject Descriptors
B.1.5 [Microcode Applications]: Microcode Applications -- *Special-purpose, Instruction set interpretation, Firmware support of operating systems/instruction sets*

General Terms
Design, Security, Algorithms, Performance.

Keywords
Microcoded coprocessor, Cryptographic biometrics, Fuzzy vault scheme, Fingerprint verification.

1. INTRODUCTION
An authentication system based on biometric information offers greater security, and is more convenient than the traditional methods of personal verification. Along with the rapid growth of this emerging technology, the system performance, including the matching accuracy and speed, is continuously improved. In a fingerprint-based biometric matching system, the comparison is made between the features extracted from an input fingerprint image and a reference template. Because of the uniqueness and sensitivity of the reference template data, secure storage is a key factor for the biometric system security. This is especially an issue for embedded applications. Special precautions must therefore be taken to protect the template from possible attacks.

A naive approach is to encrypt the template using a secret key such as a PIN. When a matching operation needs to be performed, the system decrypts the template using the PIN and then performs the biometric matching. However, this defeats the purpose of biometric devices: one tries to be independent of PIN codes entered by the user. Moreover, some dedicated attacks still could extract the secret key using a side-channel attack (SCA) [1], and in turn the template. A clean solution to this problem is to store a noninvertible transformed version, for instance a hash, of the template on the embedded device, and to perform the comparison in the transformed space. The main property of a cryptographic random hash function is that it is a one-way function, so that the output hash value will not give any information about the input [2]. Therefore, any similarity in the input will not reflect in the output hash value. For fingerprint verification, hashing is not suitable because different fingerprint scans are not exactly the same, which means that their output hash value will always be different. To address this problem, we adopt the idea of a fuzzy vault [3][4] to conduct the biometric authentication. In a fuzzy vault scheme, a transformed version of the minutiae together with a large set of noise data is stored. A suitable fuzzy vault matching algorithm then is able to distinguish between noise and input data points. In this work, we design and implement a fingerprint verification system using this novel technique. In order to construct the system efficiently and to make it reconfigurable, we build a domain-specific microcoded coprocessor, which is optimized for fuzzy vault algorithms. It can be used for a class of applications that require a fuzzy vault scheme.

This paper is organized as follows. Section 2 introduces the algorithm we adopt and the possible design approaches. Section 3 discusses the system implementation and design flow in details. Section 4 shows results and Section 5 draws the conclusions.

CODES+ISSS'05, Sept. 19-21, 2005, Jersey City, New Jersey, USA.

2. CRYPTOGRAPHIC BIOMETRICS

2.1 Application

A novel cryptographic technique called the fuzzy vault scheme has been proposed recently [3], [4]. It integrates well-known error-control coding methods and cryptographic techniques, and can be used to combine biometric authentication and encryption. The objective of this algorithm is to store biometrics data in a 'vault', a cryptographically safe store. The classic fingerprint vault construction however is based on the assumption that the fingerprint features to match are perfectly aligned – a condition that is very difficult to achieve in practice. The algorithm we adopt in our work addresses this alignment problem in a systematic way to make a complete and adaptive authentication system based on fuzzy vault [8].

As shown in Fig. 1, in order to address the security problem posed by the leakage of the stored biometric information, instead of templates, we store a machine-generated bit stream as the PIN on the device and present it as the coefficients for a Galois Field encoding polynomial in the enrollment phase. This polynomial is used to encode the minutiae template, generating the lockset of the fingerprint fuzzy vault. The next step is to add a large number of noise points to conceal this lockset. The combination of the lockset and the noise points forms the fuzzy vault. In the fingerprint matching phase, fuzzy vault unlocking needs to be performed to generate a code PIN'. Comparison of PIN and PIN' will indicate whether the matching is successful or not [8].

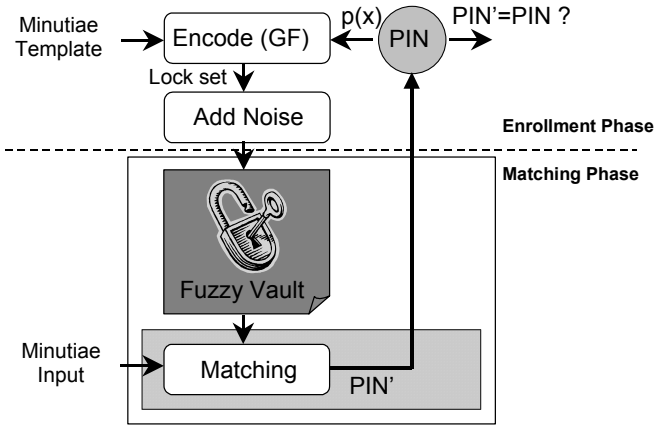

Fig. 1 Biometric fuzzy vault algorithm

2.2 Design Approach

The characteristics of fuzzy vault matching are complex decision-making as well as complex data processing. In addition, we target an implementation on a portable, resource-constrained platform. This means we need a specialized architecture, as given by one of the options of Fig 2.

A software solution based on standard program components such as a CPU can lack in execution speed, as well as in energy consumption. A full-hardware design in FPGA or ASIC, on the other hand, will achieve the required performance at the expense of flexibility and design cost. This leads to a specialized programmable solution, such as a DSP or an Application Specific Instruction Processor (ASIP). In our approach, we adopt a

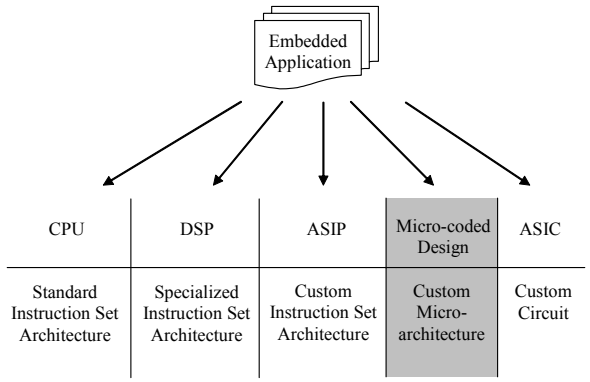

Fig. 2. Design approaches for embedded systems

microcoded coprocessor architecture, where we have full control for over all the function blocks in the datapath, the communication network, and the controller. Instead of constructing the system based on a predefined processor core, we begin from the application specifications and define our own datapth, from which a specific microcoded coprocessor called FV16 is developed. In this work, FV16 is developed in three steps: (1) Identify recurring and intensively used operations, for which special hardware modules are constructed; (2) Platform design: create interconnect, storage and control architecture to integrate datapath elements. Together with this architecture, define an instruction set, which is an abstracted version of the design; (3) Decompose the C program into assembly instructions. Generate microcode using a customized assembler.

3. IMPLEMENTATION

3.1 Architecture

The architecture of our coprocessor, FV16, is shown in Fig 3. In the fuzzy-vault algorithm, all operations execute in the $GF(2^{16})$ field. Thus all the fingerprint minutiae feature elements are represented by 16-bit integers. The fuzzy vault construction and unlocking procedure can be fully described using 16-bit arithmetic. The coprocessor is microcoded, with a separate data path and controller. This benefits the design by introducing more programmability. As shown in the figure, our system includes an ALU, a register file (RF), a data RAM, as well as a data address

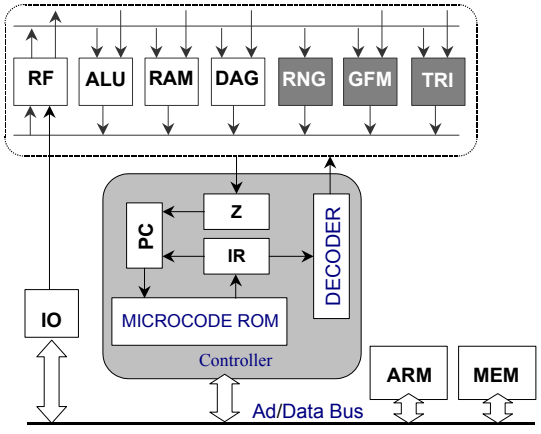

Fig. 3. Architecture for the FV16 coprocessor

generator (DAG). The ALU computation unit can perform most of the regular operations, such as addition, subtraction, bit shifting, and bit-wise logic operations. The register file contains 16 registers and each of them is 16-bit. A program counter (PC), an instruction register (IR), and a 1-bit condition register (Z) are included in the control unit. Since the PC is a 16-bit register, 64K bytes of program memory can be addressed. The Z register is used to store the result of the last compare operation. If the result was one, the Z register contains a "1"; otherwise it is "0". During execution, the FV16 coprocessor first fetches an instruction from program memory into the instruction register (IR) and sends it to the decoder. The decoded instruction is then executed. If the instruction performs a comparison operation, the condition register Z needs to be updated. For conditional branch instructions, the execution depends on the logic level of Z, which will combine with the inputs from IR to determine the proper sequence of control vectors. Therefore, each instruction requires at least three cycles to complete: fetching, decoding and executing. In order to speed up the system, a three-stage pipeline architecture is implemented. This allows a new instruction every clock cycle. For multiple-cycle instructions, such as RAM access and Galois Field multiplier, the assembler will add "NOP" instructions to avoid pipeline stalls. This simplifies the pipeline controller design.

In a high-level language programming environment, the address generation is done by the compiler, which performs variable allocations, and which converts all index expressions into integer-arithmetic operations. In the FV16 coprocessor, we implement a dedicated hardware data address generator (DAG), similar to DSP processors. Accepting a base address, this address generation unit can provides an address increase, address reset, or any particular address depending on the request. The use of such hardware address generation improves the execution performance, and eases the programming of the coprocessor.

3.2 Special Functional Block Models

Besides the blocks we discussed before, from the architecture diagram in Fig.3, there still are several function units left. These blocks are designed as special function modules to make the coprocessor more efficient in terms of speed. The special blocks are identified by means of algorithm analysis, where we find some functions are used extensively. The underlying framework of our system is Galois Field $GF(2^{16})$ arithmetic and a Galois Field multiplier is included as special computation unit. During the vault construction, a large number of randomly distributed noise points are needed to protect the biometric information. Also in the unlock procedure, the unlocking matrix includes random elements. Therefore a pseudo-random number generator is included to generate the noise required by the algorithm. In addition, a triangle block is needed for calculating the physical distance between two elements. Next we will explain these blocks individually.

3.2.1 Galois Field Multiplier

All the calculations in the fuzzy vault algorithm are based on $GF(2^{16})$ arithmetic. A Galois Field adder and a Galois Field multiplier are required. While the GF adder is implemented using logic XOR, the implementation of GF multiplication is more complicated. As shown in Fig.4, we use a bit-serial Galois Field architecture. First the shift register is initialized with all-zero state. For each clock cycle the partial product vector is added to the actual state. After 16 cycles the product is available. This entire unit we implemented as a single functional block, called the Galois Field Multiplier block (GFM). Besides using GFM, we considered two alternative implementations for Galois Field multiplication. One is to write the multiplication algorithm in C for a general purpose ARM processor and another one is programming it in assembly instructions for the FV16 coprocessor's ALU. The cycle numbers required by these three different methods are shown in Fig. 6 (GFM) in log scale. Taking advantage of the FV16 coprocessor with the GFM special block, the total execution cycles needed for the Galois Field multiplier is 85K. In contrast, it takes 1.02M on the same coprocessor without GFM block and 7.79M for C software running on the ARM. Thus, an improvement of 90 times in cycle count can be obtained using the GFM functional unit inside of FV16.

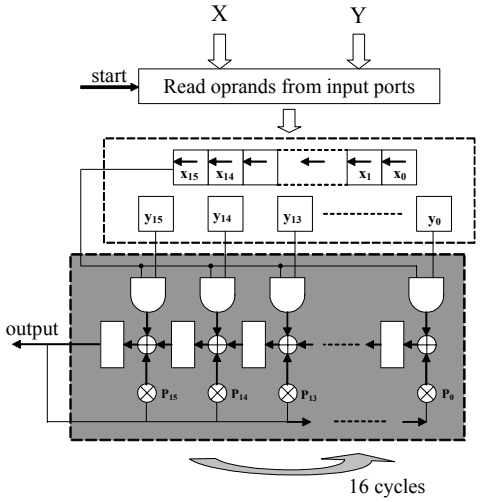

Fig. 4. Block diagram for bit-serial GF multiplier

3.2.2 Pseudo-random Number Generator

In order to provide the random elements for the vault construction and unlocking procedures, we adopt a 16-bit Linear Feedback Shift Register (LSFR) pseudo-random number generator in our design. The primitive polynomial used for minimal hardware is: $x^{16}+x^5+x^3+x^2+1$. Every clock cycle, the shift registers generate a 16bit random value. For comparison, we also implement the pseudo-random number generator in assembly code without the special hardware and in C targeting ARM, separately. The assembly code implements a LSFR in software and the C program uses the *rand()* function. Fig. 6 shows the performance results (RNG), indicating that the RNG block makes the system more than 3 times faster than the coprocessor-based design without RNG special module, and 580 times faster than the software only implementation on the ARM processor.

3.2.3 Triangle Block

At the beginning of the unlock procedure, the input value needs to be compared with the values in the fuzzy vault to find out the closest elements for constructing the unlock set. This comparison

needs to find out the physical distance between two feature points instead of a simple comparison between two numbers. According to the minutiae feature extraction procedure [8], the elements in the lock set and the unlock set are constructed by a pair of minutiae coordinates (r, θ). Thus the distance between two elements is:

$$(r_1 \times \cos\theta_1 - r_2 \times \cos\theta_2)^2 + (r_1 \times \sin\theta_1 - r_2 \times \sin\theta_2)^2$$

Since the FV16 coprocessor has only one ALU unit, trying to implement this distance function using basic instructions will take a large number of cycles. In order to speed up the system, we design a function block especially for this calculation, whose function diagram is shown in Fig. 5. Fig. 6 presents the performance improvement by implementing the triangle computation block (TRI). The system requires 10 times less clock cycles compared to those without TRI block, and 100 times less clock cycles compared to C program running on an ARM processor.

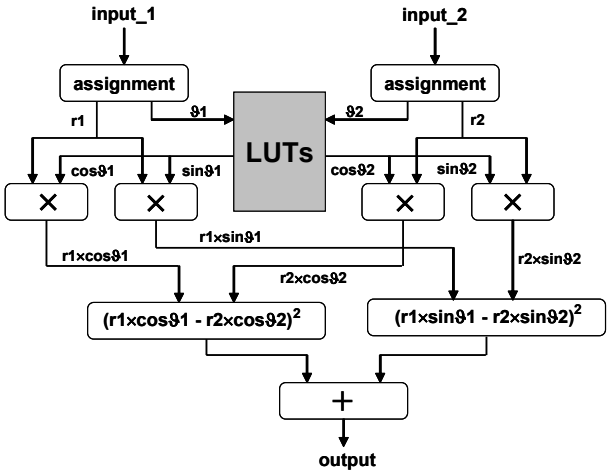

Fig. 5. Diagram for the triangle block

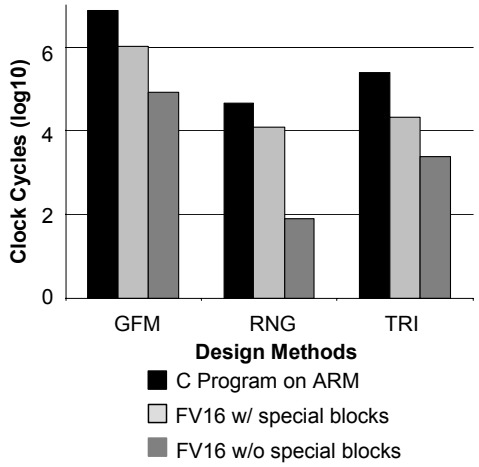

Fig. 6. Performance comparison of different design methods

3.3 Programmer's Model

Based on the architecture discussed before, we construct a programmer's model to execute all the function blocks. Each instruction is 16-bit, which in most cases is divided into three fields: the operations, the address of the source register and the address of the destination register. The operation is encoded in the first 8-bit field. Table 1 presents the instructions and their corresponding operation codes. Some special data moving instructions and branch instructions belong to special types. Their operation codes are not encoded as 8bits, and we will discuss it in detail below. All the instructions can be classified as one of the following five types:

Table 1. Instructions and corresponding operation code

8bit opcode	Instruction	8bit opcode	Instruction
0x01	ADD	0x10	DIS
0x02	SUB	0x13	DAG
0x04	XOR	0x16	MOV
0x05	LSHIFT	0x20	GETDATA
0x06	RSHIFT	0x21	WRAM
0x07	INC	0x22	RRAM
0x09	OR	0x23	DEC
0x0e	MULGF	0x24	COMP
0x0f	RNG	0x28	AND

3.3.1 Addressing Modes

There are three addressing modes for the registers of FV16 coprocessor. One is register direct addressing, which move data from one register to another. The second mode is the immediate data addressing, in which the data is contained in the instruction. Another mode is inherent addressing, where the instructions always use the same source or destination.

3.3.2 Data Transfer Operations

The coprocessor uses an internal 8K×16bit RAM block, which can be accessed with dedicated instructions. These instructions read from and write to the memory, and take care of address generation:

Mnemonics	Operation	Opcode
MOV Rn,Rm	*Rn → Rm*	0x16RnRm
MOVDi d	*d → Ri*	0b00011iiiddddddddd
GETDATA Rn	*input → Rn*	0x200Rn
RRAM Rn,Rm	*read Rn of RAM into Rm*	0x22RnRm
WRAM Rn,Rm	*Write Rn of RAM into Rm*	0x21RnRm
DAG Rn,Rm	*Depending on Rn, Data address → Rm*	0x13RnRm

3.3.3 ALU and Comparison Operations

ALU has various operations to perform the regular calculations:

Mnemonics	Operation	Opcode
ADD Rn,Rm	*Rn+Rm → Rm*	0x01RnRm
SUB Rn,Rm	*Rn-Rm → Rm*	0x02RnRm

CMP Rn,Rm	*cmp Rn&Rm, affect Z*	0x24RnRm
XOR Rn,Rm	*Rn^Rm → Rm*	0x04RnRm
LSHIFT Rn,Rm	*Rn<<8 → Rm*	0x05RnRm
RSHIFT Rn,Rm	*Rn>>8 → Rm*	0x06RnRm
INC Rn	*Rn + 1 → Rn*	0x07RnRn
DEC Rn	*Rn – 1 → Rn*	0x23RnRn
OR Rn,Rm	*Rn | Rm → Rm*	0x09RnRm
AND Rn,Rm	*Rn & Rm → Rm*	0x28RnRm

3.3.4 Branch Instructions

Since loops are used extensively in the fuzzy vault scheme, branch instructions, including non-conditional jump and conditional jump, are designed to support decision-making and control flow.

Mnemonics	Operation	Opcode
GOTO k	*goto address k in program*	0b010kkkkkkkkkkkk
CGOTO k	*on condition S, go to k*	0b011kkkkkkkkkkkk

3.3.5 Special Block Instructions

Besides the instructions described above, we also design several instructions corresponding to the three specialized function modules:

Mnemonics	Operation	Opcode
RNG Rn	*rand() → Rn*	0x0f0Rn
DIS Rn,Rm	*dis(Rn,Rm) → Rm*	0x10RnRm
MULGF Rn,Rm	*Rn * Rm → Rm*	0x0eRnRm

3.4 Design Flow

In the previous sections we discussed the architecture of the microcoded coprocessor FV16, as well as the programmer's model for writing assembly instructions. In this section, we present the system design flow, which is shown in Fig. 7. Given the application specifications, the designer will partition the C-specification in driver software running on the embedded ARM and a specialized coprocessor. Both the coprocessor architecture and the software running on it need to be designed. A specialized language, GEZEL [9], is used to construct the datapath for the coprocessor. At the same time, the C program for the algorithm, which needs to run on the coprocessor, is converted to assembly code following the programmer's model. Then the microcode is generated by the assembler and stored in the program ROM, which is part of the microcoded coprocessor. All three components, the C program for the driver, the GEZEL code for the coprocessor datapath and the microcode running on the coprocessor are co-simulated in GEZEL. GEZEL is an open design environment for domain-specific micro-architecture linked with the ARM instruction-set simulator [9]. As an example, Fig. 8 shows the GEZEL design for the Galois Field multiplier used in the FV16 coprocessor.

In order to compile the assembly instructions for the FV16 microcoded coprocessor architecture, we build a dedicated assembler based on a public available universal retargetable assembler framework from Tomasz Sztekja [10]. This is a powerful assembler and linker package, which is written fully in Java. It is very flexible and can support almost any architecture. More important, it is open source for users to port to their own processors.

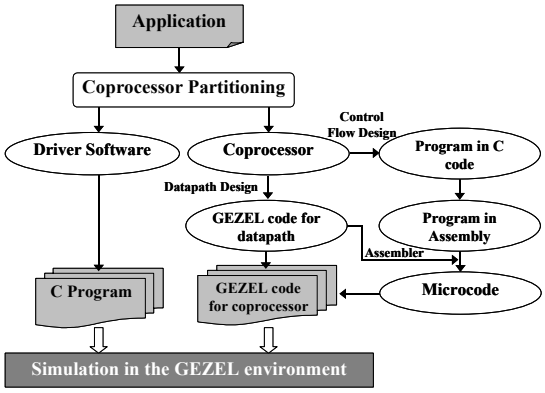

Fig. 7. Design flow for programming on the FV16 microcoded coprocessor

```
// bit-serial GF(s^16) multiplier
dp gfmul( in i1, i2 : ns(16);
          out mul: ns(16);
          in mul_st: ns(1);
          out mul_done : ns(1)) {
  reg acc, sr2, fpr, r1 : ns(16);
  reg mul_st_cmd : ns(1);
  reg reg_i1, reg_i2 : ns(16);

  sfg ini {
    fpr = 0b0000000000101101;
    reg_i1 = 0;
    reg_i2 = 0;
    acc = 0;
    mul_st_cmd = mul_st;
  }

  sfg assign_ab {
    r1 = i1;
    sr2 = i2;
    acc = 0;
  }

  sfg calc {
    sr2 = (sr2 << 1);
    acc = (acc << 1) ^ (r1 & (tc(1))
sr2[15]) ^ (fpr & (tc(1)) acc[15]);
  }

  sfg omul {
    mul = acc;
    mul_done = 1;
  }

  sfg noout {
    mul = 0;
    mul_done = 0;
  }
}
```

(a)

```
fsm gfmul_ctl(gfmul) {
  state   s1, s2, s3, s4, s5,
s6,s7,s8,s9,s10,s11,s12,s13,s14,s15,s16,s17,s18;
  initial s0;
  @s0 (ini,  noout) -> s1;
  @s1 if (mul_st_cmd) then (assign_ab, noout) -> s2;
                      else (ini, noout)  -> s1;
  @s2 (calc, noout) -> s3;
  @s3 (calc, noout) -> s4;
  @s4 (calc, noout) -> s5;
  @s5 (calc, noout) -> s6;
  @s6 (calc, noout) -> s7;
  @s7 (calc, noout) -> s8;
  @s8 (calc, noout) -> s9;
  @s9 (calc, noout) -> s10;
  @s10 (calc, noout) -> s11;
  @s11 (calc, noout) -> s12;
  @s12 (calc, noout) -> s13;
  @s13 (calc, noout) -> s14;
  @s14 (calc, noout) -> s15;
  @s15 (calc, noout) -> s16;
  @s16 (calc, noout) -> s17;
  @s17 (calc, noout) -> s18;
  @s18 (ini,  omul ) -> s1;
}
```

(b)

Fig. 8. GEZEL code for Galois Field multiplier: (a) Datapath; (b) Finite State Machine

4. RESULTS

Following the design flow, we implement the fingerprint verification algorithm based on the fuzzy vault scheme on the application specific microcoded coprocessor FV16. Using the cycle true simulation of the GEZEL, we find out the cycle number for completing the whole procedure is 166K cycles. As a comparison, we also write the embedded software in C and cross-compile it into an executable to be simulated on an ARM instruction-set simulator (ISS). The simulation shows that it takes over 13.8M cycles to finish the algorithm. In terms of source code size, 1400 lines of GEZEL code are used for the datapath description for FV16 coprocessor, and 1024 lines of assembly code are used to implement the algorithm.

After performance evaluation, the secure vault fingerprint verification system based on the FV16 coprocessor can be converted into synthesizable VHDL and run on a reconfigurable FPGA platform. The Synplicity tool Synplify Pro is used to perform the synthesis using Xilinx Virtex2 XC2V1000 as the target platform. The system results are shown in Table 2:

Table 2. Results of FV16 coprocessor

Parameters	Result	Parameters	Result
GEZEL code for coprocessor	1400 lines	Total LUTs	2960
Microcode	1024 lines	Block RAMs	16
Total cycle	166K	Critical Path	22.231 ns

5. CONCLUSIONS

We design an application specific microcoded coprocessor called FV16, based on which a HW/SW co-design for a secure biometric authentication system is constructed. An instruction set, as well as the programmer's model, is constructed for writing assembly programs targeting on this architecture. In this work we propose a complete design flow to show how the design tasks are integrated. From the design flow it is clear how other applications can be mapped onto this microcoded coprocessor. Also the results show that using our coprocessor makes the design over 83 times more efficient compared to software only implementations.

6. ACKNOWLEDGMENTS

The authors would like to acknowledge the funding of NSF account no CCR-0310527 and UC MICRO.

7. REFERENCES

[1] Tiri, K., and Verbauwhede, I, A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation, Design, Automation and Test in Europe Conference, pp. 246-251, February 2004.

[2] Anderson, R. J., Security Engineering, A Guide to Building Dependable Distributed Systems, John Wiley & Sons, 2001.

[3] Juels, A. and Sudan, M., A fuzzy vault scheme, Proceedings 2002 IEEE International Symposium on Information Theory, pp.408. Piscataway, NJ.

[4] Juels, A. and Wattenberg, M., A fuzzy commitment scheme, 6th ACM Conference on Computer and Communications Security, 1999, pp.28-36, New York, NY.

[5] Keutzer, K., Malik, S., and Newton, A.R., From ASIC to ASIP: the next design discontinuity, 2002 IEEE International Conference on Computer Design, 2002, pp.84-90. Los Alamitos, CA.

[6] www.tensilica.com

[7] Clancy, T.C., Kiyavash, N., and Lin, D.J., Secure smartcard-based fingerprint authentication, ACM Workshop on Biometrics: Methods and Applications, Nov. 2003, pp. 45-52, Berkeley, CA.

[8] Yang, S. and Verbauwhede, I., Automatic Secure Fingerprint Verification System Based on Fuzzy Vault Scheme, 2005 IEEE International Conference on Acoustics, Speech, and Signal Processing, pp609-612, March 2005, Philadelphia, PA.

[9] Schaumont, P. and Verbauwhede, I., Domain-specific tools and methods for application in security processor design, Kluwer Journal for Design Automation of Embedded Systems, pp. 365-383, November 2002.

[10] Tomasz Sztejka, Universal Retargetable Assembler, http://www.dornet.pl/~sztejka/sztejkat/Retargetable_assembler.html.