Hardware-Software Co-Design of an Iris Recognition Algorithm

Mariano López* (contact author): Technical University of Catalonia, Avda. Victor Balaguer, s/n, 08800 Vilanova i Geltrú, Spain, Phone: +34938967737; fax: +34938967700, e-mail: lopezg@eel.upc.edu.

John Daugman: University of Cambridge, Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK., e-mail: john.daugman@cl.cam.ac.uk.

Enrique Cantó: Rovira Virgili University, Av. Països Catalans 26, 43007 Tarragona, Spain, e-mail: enrique.canto@urv.net.

Abstract— This paper describes the implementation of an iris recognition algorithm based on hardware-software co-design. The system architecture consists of a general-purpose 32-bit microprocessor and several slave coprocessors that accelerate the most intensive calculations. The whole iris recognition algorithm has been implemented on a low-cost Spartan 3 FPGA, achieving significant reduction in execution time when compared to a conventional software-based application. Experimental results show that with a clock speed of 40 MHz, an IrisCode is obtained in less than 523 ms from an image of 640x480 pixels, which is just 20% of the total time needed by a software solution running on the same microprocessor embedded in the architecture.

1. INTRODUCCTION

Recent years have seen the growth of new application domains for image processing and pattern recognition in the field of automated human identification, for security purposes and for logical or physical access control, based on personal biometric characteristics. Authentication systems based on biometrics determine the user's identity on the principle that some physiological or behavioral characteristics are unique for each person, and are more tightly bound to a person than a token object or a secret, which can be lost or transferred. Automated real-time biometric systems such as fingerprint or iris recognition have been successfully deployed in several large-scale public applications, increasing reliability and convenience for users, and reducing identity fraud. Usually the implementation of biometric algorithms is carried out using high-performance microprocessors working at clock frequencies in the GHz range. These devices are designed with an advanced architecture based on several pipeline stages, cache memory, high-speed communication buses and additional units that facilitate rapid execution of complex algorithms. On an Intel Pentium 4 at 3.2GHz, with 1GB of RAM memory, the average execution time of a fingerprint recognition algorithm, including enrollment and matching, is about 778 ms and on a similar microprocessor the computing time for iris image analysis and creation of an IrisCode is about 30 ms [1][2]. However, such software implementations could restrict the application of biometrics to specific markets due to the microprocessor cost.

Devices available in the low-cost consumer market are generally too slow for applications requiring intensive computations. For example, an iris recognition algorithm running on an ARM922T at 160MHz executes in 3162 ms, which is about 80 times slower than the execution of the same code on a high-performance microprocessor. The use of dedicated hardware is an alternative for implementing operations that require high-speed parallel processing [3]-[12]. Additionally, outstanding results can be achieved if the structure of the algorithm allows the hardware to employ several pipeline stages. For example, under certain conditions, an image enhancement routine usually employed in a fingerprint recognition algorithm can be processed in dedicated hardware faster than on a Pentium clocked at a frequency 30 times higher [7]. However, designing such a hardware solution is less justifiable for algorithms requiring floating point computations or when sequential operations hinder the application of pipeline and parallelism. In these cases, the area and the effort devoted to design the system might not be justified by the benefits gained.

Architectures based on hardware-software co-design combine the advantages of both hardware and software solutions. Such systems contain an embedded microprocessor and several dedicated hardware units connected via a communication bus. By offloading processor intensive tasks to dedicated hardware and assigning operations that require high-speed serial processing to the microprocessor, the performance and cost of the whole system are substantially improved. For instance, this methodology has been successfully applied to designing a biometric fingerprint verification system. That architecture, implemented on a Virtex II FPGA, is composed of a general purpose fixed-point processor and a DFT (Discrete Fourier Transform) hardware accelerator used to determine the dominant ridge flow direction. Results show that the coprocessor permits a 55% and 60% execution time reduction for the minutiae extraction and matching, respectively [8]-[10]. Other publications show similar improvements when dedicated hardware units are used in order to implement different fingerprint algorithms or systems based on other biometric modalities such as face or speaker recognition [5][11][12].

The purpose of this paper is to describe an implementation of an iris recognition algorithm based on a hardware-software co-design methodology, suitable for integration either in ASIC (Application Specific Integrated Circuit) or FPGA. The experimental results reported in this paper were obtained using a low-cost Spartan-3 FPGA clocked at 40 MHz.

This paper is organized in 5 sections. Section 2 reviews briefly the basic principles underlying the iris recognition algorithm. Section 3 analyzes the functions involved in the algorithm, assessing which ones are suitable to be executed on the microprocessor and which ones should be implemented in dedicated hardware. Section 4 describes the internal structure of the embedded system, and finally Section 5 presents the experimental results.

2. ALGORITHM REVIEW

The implemented iris recognition system is based on the algorithms developed by Daugman, which are documented in [13]-[15]. These algorithms are the basis of all currently deployed iris recognition systems and they will be only briefly reviewed here.

The iris image is acquired usually within a distance of about 50 cm by a camera using infrared light in the 700nm-900nm band and resolving about 100-200 pixels in iris diameter. Specular reflections of the illumination on the cornea or eyeglasses are detected and removed, and the boundaries of the iris are determined. Fig. 1 illustrates such a captured iris image, with overlaid graphics showing the automated detection of the iris inner and outer boundaries as well as its eyelid occlusion boundaries. The centers and radii of iris and pupil are approximated initially by applying an integrodifferential operator that behaves as a circular edge detector:

$$max_{(r,xo,yo)} G_\sigma(r) * \frac{\partial}{\partial r} \oint_{r,xo,yo} \frac{I(x,y)}{2\pi\pi} ds \tag{1}$$

where *I(x,y)* is the image that contains the eye, symbol * denotes convolution and $G_\sigma(r)$ is a smoothing function of scale *σ*. The operator is applied iteratively in a multi-scale, coarse-to-fine strategy to converge rapidly on estimates of the three parameters of each circular model. Then, the upper and lower eyelid boundaries may be described as quadratic or cubic splines, whose parameters are estimated by statistical model-fitting techniques.

Following the initial approximation of the iris and pupil boundaries as circles for purposes of iris localization, their actual shapes are recalculated using active contour models [2]. This finer analysis allows a more precise description of these boundaries which are often significantly non-circular (see Fig. 2). The box in the lower-left corner of Fig. 2 shows curvature maps for the inner and outer iris boundaries. Dotted curves in the box and on the iris are Fourier series approximations to the actual boundaries, enabling a flexible and appropriate coordinate system to be embedded in the iris.

The defined region between the inner and outer boundary of the iris is normalized into a doubly-dimensionless, not necessarily concentric, pseudo-polar coordinate system (r,θ), where r lies in the unit interval **[0,1]** and θ is a cyclic angular variable over **[0,2π]**. This mapping normalizes the iris and compensates for deformations caused by pupil dilation or constriction. This mapping also achieves invariance to the user distance from the camera and to the position of the eye. Let $(x_p(\theta), y_p(\theta))$ and $(x_s(\theta), y_s(\theta))$ be the set of points corresponding to the pupil and limbus (outer) boundaries. The generalized, non-concentric coordinate system can be described as the following linear combination:

$$\begin{bmatrix} \mathbf{x(r,\theta)} \\ \mathbf{y(r,\theta)} \end{bmatrix} = \begin{bmatrix} \mathbf{x_p(\theta)} & \mathbf{x_s(\theta)} \\ \mathbf{y_p(\theta)} & \mathbf{y_s(\theta)} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{1-r} \\ \mathbf{r} \end{bmatrix} \tag{2}$$

Once the iris has been segmented from the image and mapped into normalized dimensionless coordinates, the iris texture is encoded into an IrisCode through a process of demodulation that extracts phase sequences. The IrisCode contains 2048 data bits, derived by projecting local regions of the iris onto quadrature 2-D Gabor wavelets. Additionally, a mask array of the same size is computed to mark those bits obscured by eyelids, eyelash occlusions detected by statistical inference, or corneal reflections. After this encoding stage, the iris template can be stored or matched against a database that contains previously enrolled templates. The matching engine is based on computing the Hamming distance (fraction of disagreeing bits) between two different IrisCodes gated by their associated mask vectors. This search engine mainly performs simple Boolean operations using XOR and AND gates that can be implemented with large bit-wise parallelism either on a microprocessor or in dedicated hardware.

The algorithm discussed in this paper has been tested on a database of 632,500 different iris images, leading to 200 billion pair comparisons proving extreme robustness against false matches [2][13]. For example, with a Hamming distance threshold of 0.30 the observed false match rate was 1 in 8 million, and at a threshold of about 0.25 the false match rate was 0 in 200 billion.

3. PROFILING AND HARDWARE-SOFTWARE PARTITIONING

3.1 Architecture description

Fig. 3 shows the generic architecture of a system based on hardware-software co-design. The system consists of a microprocessor acting as master that manages the organized execution of a program, the communication between input/output devices and the control of information through the system's buses [16].

Hardware coprocessors cooperate actively with the software application executed on the microprocessor and are designed to provide a specific functionality regarding some part of the application. The degree of complexity of the functionality depends on the architecture partitioning level, which can range from simple operations or instructions (fine granularity) to complex processes related to functions or routines (coarse granularity). In any case, the partitioning task consists in determining which parts of the system are best suited for execution by software or for synthesis in dedicated hardware, in order to satisfy a set of constraints and goals such as performances, cost or area.

Partitioning can be considered from two different viewpoints. A software-oriented approach initially considers the whole application as running on the microprocessor. In the partitioning process, parts of the software application whose sizes depend on the granularity of the partitioning are moved to hardware until constraints are met. Conversely, in a hardware-oriented approach the migration is done in the reverse direction, from hardware to software [17]. In our particular case, given that the whole algorithm exists in ANSI C, the partitioning process is undertaken with a software-oriented approach. Since the arithmetic operations (integer or float) and the programming structure at block and control level are different in each function, a fine-grained partitioning would entail designing a large number of dedicated hardware units, which requires major design effort and excessive area for system implementation. The hardware-software partitioning discussed in this paper is carried out at function level. The advantage of a coarse-grained partitioning is that it requires a small number of hardware coprocessors and reduces communication delays.

3.2 Design criteria for hardware-software partitioning

Determining which functions described in section 2 are most suited for hardware implementation depends on the following design criteria:

- Time needed by the microprocessor to execute a function as a percentage of the execution time of the whole algorithm.

- Hardware speed-up factor (acceleration), defined as the ratio of execution times of software to hardware implementations.

- Complexity of hardware design and need to incorporate specific IP cores (reusable units of logic used as building blocks in FPGA or ASIC designs) for certain arithmetic operations.

The percentage of the total execution time consumed by each function was directly calculated running the algorithm on four different processors. The first one is an Intel Centrino 1.7 GHz, a high-performance microprocessor suitable for use in applications that manage a database with hundred of thousands of users (e.g. airport check-points), and where the microprocessor cost and its power consumption are not relevant factors. The second profile was obtained using a 32-bit ARM922T at 160 MHz, a medium-performance microprocessor whose architecture is the most widely employed in consumer electronics such as mobile phones and PDAs [18]. The third microprocessor chosen to evaluate the execution times was an Intel Pentium at 133 MHz. This device is currently used by the PIER handheld camera, a commercial portable device for iris recognition developed by Securimetrics [19]. The last benchmark was obtained using Microblaze at 40 MHz, a soft-core microprocessor developed by Xilinx suitable for designing embedded systems, allowing easy connection of custom coprocessors [20].

On the other hand, predicting the acceleration ratio (speed-up factor) between software and hardware implementations is an elaborate process. In order to illustrate the process that we proposed in this paper for estimating the execution time of a function implemented in hardware, let us consider the simple piece of code shown in Fig. 4.a. (For clarity, Fig. 4.b shows a semi-unrolled version of the same code). This example code is a nested loop that calculates the variance of an image of size NxM pixels (8-bit grey-level). The hardware implementation of this code basically requires several adders, one multiplier and a memory controller that manages access to the image located in external SRAM memory (usually the size of an image requires that it resides outside the FPGA). Any operation can be carried out in one clock cycle, except memory reading which needs two cycles. We assume that hardware is configured with only one external memory. Under this assumption the most critical resource in terms of time consumption is the memory access (implemented by means of a memory controller) that limits the best timing performance to 2*N*M cycles.

The key to achieving execution time as near as possible to this limit is to design pipelined hardware that prioritizes memory access. In Fig. 4.b, the inner loop associated with index j contains six operations that are distributed in two groups (the number of groups coincides with the latency in cycles of the critical resource). The operations involved in each group are calculated in parallel, according to the scheduling scheme shown in Fig. 5. Clearly the approximate hardware execution time is 2*N*M cycles.

The method of estimating execution time shown in this example can be formulated in a general way as follows:

- Given a function, finding its most time-consuming operation. The total time consumed by this resource is equal to the number of times that the resource is used multiplied by its associated latency (number of cycles needed by the resource to give a result).
- When several independent loops are present in the function, the total time consumed is the sum of their individual time consumptions.
- Coprocessors are able to read and write directly in external SRAM, by means of a memory controller that allows one access every two clock cycles (only one memory controller is available).
- Additions, subtractions and multiplications involving integer operands are carried out in one clock cycle (multiplications are implemented by using the specific internal hardware multipliers of the FPGA).
- Integer division requires a number of clock cycles equal to the bit length of the operands. Unlike multiplication, this block cannot be directly synthesized on the FPGA and requires design of a specific arithmetic unit for its implementation.
- The number of clock cycles required for floating point operations is given in Table I.

Latencies and hardware resources presented in this table have been obtained from the single-precision floating point core designed by Xilinx [21]. Note as the multiplication requires less cycles and area than the addition due to the internal hardware multipliers available on Spartan 3.

TABLE I

AREA AND NUMBER OF CYCLES FOR OPERATIONS WITH SINGLE-PRECISION FORMAT ON SPARTAN 3E FPGA

Floating-point operation	Cycles	LUTs (Look-up table)	FFs (Flip-flops)
Add/Sub	13	580	591
Multiplication	6	185	275
Division	28	234	229
Sqrt (square root)	28	214	206
Fixed to float conversion	6	221	227
Float to fixed conversion	5	251	237

3.3 Hardware-software partitioning based on Microblaze

Tables II and III show the execution times for the microprocessors detailed in section 3.2, for the enrollment and matching processes, for verification and identification mode, respectively, as well as the percentage that each function represents of the total execution time of the algorithm. These results have been obtained using images with 640x480 pixels. Note that these results depend strongly on the specific architecture of each microprocessor, the presence of a floating point unit (FPU), the distribution of the executable code between on-chip and external memory and the functions and parameters used to solve the algorithm. The profiler provides execution times for each function, but only data obtained with Microblaze are considered for hardware-software partitioning, since this is the microprocessor actually used to build the embedded system.

Intel Centrino is by far the fastest microprocessor, executing all the functions in less than 40 ms. The execution time on ARM922T is 3162 ms, which is about 18% slower than Microblaze working at a clock frequency four times lower. Note as the ARM922T architecture used in this paper lacks a floating point unit (basically to save on power and area when using ARM in low-cost applications), unlike more advanced architectures in the same family [18]. This fact has a significant influence on the execution time of those functions using floating-point operations.

TABLE II

EXECUTION SPEEDS OF VARIOUS FUNCTIONS IN THE IRIS RECOGNITION ALGORITHM RUNNING ON FOUR DIFFERENT MICROPROCESSORS

Function name	Time / % percentage			
	Centrino 1.7GHZ 1MB Cache External 512 MB DDRAM	ARM922T 32-bit 160MHz On-chip 40kB BRAM External 32MB SDRAM	Pentium 133MHz 8kB+8kB Cache External 65 MB SDRAM	Microblaze 32-bit 40MHz On-chip 64kB BRAM External 2MB SRAM
Scrub specular reflections	4,5 ms/ 11.4%	85 ms / 2.7%	117 ms / 10.5%	334 ms / 12.9%
Localize iris	14.4 ms / 36.6%	670 ms / 21.2%	447 ms / 40.2%	1157 ms / 44.7%
Localize pupil boundary	6.4 ms / 16.3%	280 ms / 8.8%	198 ms / 17.8%	369 ms / 14.2%
Detect and fit eyelids	1.8 ms/ 4.6%	95 ms / 3.1%	34 ms / 3.1%	113 ms / 4.4%
Fine-tune models of iris inner & outer boundaries	7.3 ms / 18.6%	1870 ms / 59.1%	210 ms / 18.9%	440 ms / 17.0%
Dimensionless sampling	0.85 ms / 2.2%	10 ms / 0.32%	37 ms / 3.3%	43 ms / 1.6%
Remove eyelashes	0.75 ms / 1.9%	12 ms / 0.38%	23 ms / 2.1%	32 ms / 1.2%
Create IrisCode	3.3 ms / 8.4%	140 ms / 4.4%	46 ms / 4.1%	103 ms / 4.0%
Overall algorithm	**39.3 ms / 100%**	**3162 ms / 100%**	**1112 ms / 100%**	**2591 ms / 100%**

TABLE III

EXECUTION TIMES OF THE MATCHING ENGINE RUNNING ON FOUR DIFFERENT MICROPROCESSORS FOR VERIFICATION AND IDENTIFICATION AGAINST A DATABASE OF 512 IRISCODES

Function name	Time				Hw/Sw
	Centrino 1.7GHZ	ARM922T 160MHz	Pentium 133MHZ	Microblaze 40MHz	
Verification	60 µs	580 µs	535 µs	1.32 ms	Sw
Identification	2.4 ms	68 ms	61 ms	131.7 ms	Sw

The ISE design suite 10.1, the Xilinx software used to build the system, allows the integration of Microblaze 7.10.d configured with an IEEE-754 compatible single-precision floating point unit. In contrast to previous versions of the same microprocessor, its FPU allows a cast between float and integer signed type (and vice versa) by means of a single assembler instruction. These and other floating point operations are frequently used in some functions of the iris algorithm, which permits Microblaze to compensate for its much lower clock frequency by more efficient execution of these computations. As we will see later in section 4, to speed-up the execution time we have designed our own memory controller to access the external SRAM memory. This memory controller prevent the configuration of Microblaze with cache memory that is only compatible when using a Multi-Cannel OPB controller specifically designed by Xilinx. On the other hand, the Pentium clocked at 133 MHz executes the overall algorithm in 1112 ms. Note that the ratio between the clock frequency of this microprocessor and Microblaze is 3.325, but the execution is only 2.33 times faster.

Table IV presents the execution time of a potential hardware implementation using our estimation method and the acceleration ratio between software and dedicated hardware based on the profiler provided with Microblaze. The accuracy needed in some operations, and primarily the wide dynamic range of some variables used in some functions shown in Table II, force such variables to be defined as floats. The hardware design of these functions requires incorporating a dedicated floating point unit different from the FPU available on the microprocessor. There are some vendors of IP cores that provide reliable designs of FPUs easily adaptable to particular custom requirements. This option simplifies the design effort but generally is only valid for a specific technology related to a FPGA family and manufacturer. Another possibility is to design our own floating point unit, usually at low-cost but adding a major complexity and offering poorer performance. An additional factor to be considered is the extra area needed for inclusion of this core. In the floating point unit presented in Table I, this area basically depends on the latency associated with the operations and the bit width of the operands, occupying approximately 840 CLB slices. As the next section will show, this result is similar to the area needed by the iris and pupil localization coprocessors. Due to these disadvantages, generally the inclusion of a floating point unit as part of a coprocessor is only justifiable when functions represent a high percentage of the total execution time and when such a design substantially improves the acceleration ratio between software and hardware implementations.

The subroutines to remove eyelashes, to detect and fit eyelids, to perform dimensionless sampling and to create the IrisCode, are executed on the microprocessor. Note that these functions mainly contain floating point operations, they represent a reduced percentage of the total execution time (each one less than 5%) and their theoretical acceleration ratio is fairly low (less than 6 as table IV shows). Conversely, the subroutines to scrub specular reflections and to localize the iris and pupil boundaries are suitable for hardware implementation. These functions contribute substantially to the total execution time (together they represent about the 71% of the total time), they are based on integer arithmetic operations and they have an acceleration ratio higher than 11 compared with their software execution.

TABLE IV

CRITICAL RESOURCE, ESTIMATION TIME, HARDWARE ACCELERATION AND HW/SW PARTITIONING

Function name		Critical resource	Operations involved in the function (Integer or/and floating)	Estimated time	Acceleration ratio	Hw/Sw
Scrub specular reflections		Mem. Access	Int: add,div,mult	18.9 ms	17.7	Hw
Localize iris		Mem. Access	Int: add,mult	69.5 ms	16.6	Hw
Localize pupil boundary		Mem. Access	Int: add,div,mult	32.5 ms	11.3	Hw
Detect and fit eyelids		Mem. access + floating point	Int: add,mult Float: add,div,mult,sqrt	24.9 ms	4.5	Sw
Fine-tune iris inner and outer boundaries (active contours)	Integer	Mem. Access	Int: add,div	35.8 ms	10.2	Hw
	Floating point	Floating point	Float: add,div,mult	56.9 ms	1.2	Sw
Dimensionless sampling		Mem. access + floating point	Int: add,mult	7.9 ms	5.4	Sw
Remove eyelashes		Mem. access + floating point	Int: add,mult Float: add,div,mult	5.4 ms	5.9	Sw
Create IrisCode		Mem. access + floating point	Int: add,mult,div Float: add,div,mult	28.7 ms	3.6	Sw

The active contours function (fine-tuning of the inner and outer iris boundaries) presents a significant computational cost when executed on the Microblaze microprocessor. However, its implementation requires a floating point unit. A deeper analysis shows that this function can be divided into two different parts. The first one contains only integer operations whereas the second one mainly uses floating point computations. On the other hand, their execution times as a percentage of the global function are significantly different (83.5% and 16.5% for the integer and floating part, respectively). Likewise, the procedure for estimating execution time reveals important differences in the acceleration ratio, which is about 10.2 times and 1.2 times for the integer and floating point parts, respectively. Thus, for this function with these particular features we propose a partitioning into two blocks with a hardware implementation for the integer part and a software execution for the floating point part. Table IV summarizes the proposed hardware-software partitioning of the whole IrisCode algorithm (enrollment phase).

The execution time of the search engine (the matching phase) depends on the size of the database that contains N IrisCodes associated with N different iris images. In a verification process (N=1), whose aim is to confirm or deny a particular asserted identity, the execution time for the matcher on the 40 MHz Microblaze is 1.32 ms (genuine user). In an identification process for recognizing a person by exhaustively searching a list of previously enrolled users, the execution time on the same device is 131.7 ms for a search database containing 512 IrisCodes. In identification mode (searching the whole database), IrisCode bytes are undersampled to speed-up the comparisons by pre-qualifying only good candidate matches for fully detailed comparison. This speed-up is important in large databases because the IrisCode comparisons must be done in each of many orientations, usually 21 rotations, since the actual tilt angles of heads and eyes are not known in advance. It is appropriate for the search and matching function to be executed by software since it is already so efficient, but if databases grew to the scale of national populations then clearly a dedicated hardware implementation would be appropriate.

4. HARDWARE DESING

The overall internal structure of the system is depicted in Fig. 6. Microblaze accesses data and instructions by means of a dedicated bus called LMB (Local Machine Bus), which connects the microprocessor to an internal dual-port 64Kbyte RAM memory that requires at least two clock cycles for reads and for data writes. The limited size of on-chip memory forces long arrays and images to reside in external 2Mbyte SRAM, which is connected through a memory controller to an OPB bus (On-chip Peripheral Bus). This bus is based on the standard CoreConnect of IBM, whose speed is limited by off-chip memory access delays or bus arbitration overheads resulting in five to seven clock cycles per read.

The external SRAM memory is a common resource shared by the microprocessor and the coprocessors. The system was designed so that the coprocessors have direct access to external memory, without requiring use of the OPB bus. This design is more efficient, since coprocessors can read and write faster (2 clock cycles) and thereby avoid delays due to bus communication overheads. Since we adopted a coarse-grained hardware-software partitioning, these overheads are mainly due to the communication established between Microblaze and the four dedicated coprocessors, which can be considered negligible compared to the total execution time.

Any peripheral connected to the OPB bus has associated a memory space ranging from a base address, which establishes its lower bound, to a high address upper bound. The system architecture has been designed in such a way that coprocessors and RAM memory share the same memory space. This feature requires an arbitration mechanism that allows the microprocessor to establish a bidirectional communication with SRAM memory and coprocessors. This mechanism must consider the following scenarios:

- The microprocessor reads or writes to external memory and requires control of the input lines of the memory controller.
- The microprocessor sends information to coprocessors, such as image pointers or data, necessary for executing a function implemented in hardware.
- The coprocessor is activated by means of a signal generated by the microprocessor through the OPB bus.

- An activated coprocessor takes control of the memory controller in order to read or write in external SRAM memory.
- The coprocessor finishes execution of a function and the microprocessor reads the returned information through the OPB bus.

The input lines of the memory controller are managed by a multiplexer that assigns their control depending on signal select, which is generated by means of a decoder that considers the OPB bus signals read/write and address. The first four positions in memory are reserved as registers for exchanging information between microprocessor and coprocessors. As the decoder truth table of Fig. 7 shows, when the microprocessor writes on SRAM memory ($\mathrm{OPB\_\overline{R}/W}$ and SRAM_select have value 1) the multiplexer directly connects the controller input lines to the OPB bus. In this working mode, the microprocessor takes control of memory and is able to transmit information to the coprocessors using the reserved memory positions, or to write to the rest of memory. Moreover, when the microprocessor reads from memory ($\mathrm{OPB\_\overline{R}/W}$ and SRAM_select take value 0 and 1, respectively) two different situations can arise. If the microprocessor addresses one of the reserved memory positions (signal Copro_select equal to 1), the two least significant bits of the address line determine the coprocessor that must be activated. Once the calculation is finished the result is placed in the proper reserved memory position and is read and used as a parameter for subsequent stages. In contrast, if a non-reserved address is selected (signal Copro_select equal to 0), the multiplexer assigns the input lines of the controller to the microprocessor so that it can read from external memory.

5. EXPERIMENTAL RESULTS

The coprocessors were defined in the VHDL high-level description language and were implemented using the EDK (Embedded development kit) software package of Xilinx. Experimental results were obtained with the AVNET hardware development board that contains a Xilinx FPGA Spartan 3 XC3S2000, 2MB of SRAM memory and several communication peripherals [22].

Table V presents the area occupied by each coprocessor and its maximum clock frequency due to the critical path. These results were obtained using the Leonardo Spectrum synthesis tool, selecting a Spartan 3 FPGA [23]. Table VI shows the execution times of functions whether implemented by software or hardware, given a clock frequency of 40MHz. The system needs 522.6 ms to culminate in an IrisCode, which is about 5 times faster than the software-only solution presented in Table II. Since the clock frequency is 40MHz, the iris code is created in 20.9 Megacycles. As Table II shows, the same processing is done using an Intel Centrino at 1.7 GHz in 66.81 Megacycles, which is about 3.2 times the number of cycles needed by our system. On the other hand, our implementation is 2.12 faster than the execution of the whole algorithm on a Pentium 133 MHz, the current microprocessor used by the PIER handheld camera developed by Securimetrics.

The capture of the iris by most cameras is a process that requires active user cooperation in order to obtain an good quality image. The average time needed to position an eye in the right place and distance from the camera is typically about 2 seconds. Consequently, using our proposed implementation the time needed to create the IrisCode is less than 27% of the total time for the identification process.

TABLE V

AREA AND MAXIMUM CLOCK SPEED OF EACH COPROCESSOR

Function name	Area occupied (CLB slices)	Maximum clock speed
Scrub specularities	453	80.7 MHz
Localize iris	797	89.3 MHz
Localize pupil boundary	981	69.7 MHz
Active contours (hardware)	1342	70.9 MHz

Table VI also shows that the acceleration ratios (comparing software and hardware implementations) are in all cases greater than a factor of 10, which represents an important improvement and a substantial reduction of the processing time. The function to scrub specular reflection achieves the greatest ratio (maximum speed-up), almost 17 times faster than its software execution. The functions iris and pupil location have also a significant acceleration ratio of 15.5 and 10.9, respectively. Moreover, the function active contours also achieves an important acceleration. As noted previously, this subroutine uses a mixed hardware-software solution with part of the function running on the microprocessor and the rest implemented in a dedicated coprocessor. The software part has an execution time of 72.6 ms, whereas the coprocessor needs only 36 ms for the operations that were the most computationally expensive in the hardware-software partitioning analysis.

TABLE VI

EXECUTION TIMES FOR SOFTWARE FUNCTIONS AND HARDWARE COPROCESSORS AND ACCELERATION RATIO WORKING AT A CLOCK FREQUENCY OF 40MHZ

Function name	Hardware/software execution time	Acceleration
Scrub specular reflections	19.6 ms (Hw)	17.04
Localize iris	69.8 ms (Hw)	15.57
Localize pupil boundary	33.6 ms (Hw)	10.98
Detect and fit eyelids	113 ms (Sw)	--
Active contours to fit iris inner & outer boundaries	36ms (Hw) +72.6 ms (Sw) =108.6 ms	10.20
Dimensionless sampling	43 ms (Hw)	--
Remove eyelashes	32 ms (Sw)	--
Create IrisCode	103 ms (Sw)	--
Overall algorithm	**522.6 ms**	

TABLE VII

ACTUAL AND ESTIMATED HARDWARE EXECUTION TIMES AND PERCENT VARIANCE

Function name	Actual execution time	Estimated time	% Error
Scrub specular reflections	19.6 ms	18.9 ms	3.5%
Localize iris	69.8 ms	69.5 ms	0.4%
Localize pupil boundary	33.6 ms	32.5 ms	3.2%
Active contours to fit iris inner & outer boundaries	36ms	35.8 ms	0.5%

Table VII shows actual execution times of those functions implemented by hardware and also their execution times as predicted by the estimation procedure presented in section 3. The Table also shows the percent variance between the speeds in order to document the accuracy of the estimation method. Note that the maximum value for this error is less than 4%, with the most accurate estimate erring by only 0.4%.

6. CONCLUSIONS

Low cost and rapid response times are important parameters for practical authentication systems. Usually developers of biometric algorithms assume that hardware platforms have enough computational capability to execute the algorithms with acceptable speed, using in most cases high-performance and high-cost microprocessors. The main purpose of the work described in this paper was to implement an iris recognition algorithm using a low-cost FPGA. The design methodology was based on a hardware-software co-design, as a viable alternative to the traditional approach based only on software.

The system architecture consists of a 32-bit general purpose microprocessor and several dedicated hardware units. The microprocessor executes in software the less computationally intensive tasks, whereas the coprocessors speed-up the functions that have higher computational cost. Design criteria for hardware-software partitioning were based on the profiler, the proposed method for estimating the hardware execution time, and the need to incorporate specific cores. The simple method proposed in this paper for estimating the execution time of a function, gives beforehand a valuable information to decide about the convenience of its implementation in hardware. Depending on the function implemented, the designed coprocessors speed-up the processing time in all cases by a factor greater than 10 compared to its software execution. The profiler was obtained by executing the iris algorithm on four microprocessors, with different performances and features such as clock frequency, internal and external memory or floating-point unit availability. Except for the high-performance microprocessor, the execution time was in all cases above 1100 ms with a maximum operating frequency of 160 MHz.

The proposed hardware-software co-design was implemented on a low-cost Spartan 3 FPGA. Results show that with a clock frequency of 40MHz the system is able to execute the entire iris recognition algorithm in 522.6 ms from an image of 640*480 pixels. The best exclusively software solution implemented on a microprocessor as Pentium 133 MHz gave an execution time of 1112 ms, which is about 2.12 times slower than our system operating at a clock frequency 3.3 times lower.

REFERENCES

[1] Fingerprint Verification Competition 2006 (FVC2006). Available: http://bias.csr.unibo.it/fvc

[2] Daugman, J. G.: "Probing the uniqueness and randomness of IrisCodes: Results from 200 billion iris pair comparisons," Proceedings of the IEEE, vol. 94, no. 11, Nov. 2006, pp 1927-1935.

[3] Ratha N., Rover D. and Jain A. K.: "An FPGA-Based point pattern matching processor with application to fingerprint matching," CAMP '95, Italy, 1995, pp. 394-401.

[4] Ratha N., Rover D. and Jain A. K.: "Fingerprint Matching on Splash 2," FPGAs in a Custom Computing Machine, D. Buell, J. Arnold and W Kleinfolder (eds.) IEEE Computer Society Press, 1996, pp. 117-140.

[5] Cantó E., Canyellas N., Fons M., Fons F., López M.: "FPGA Implementation of the Ridge Line Following Fingerprint Algorithm," Proceedings of the 14th International Conference on Field-Programmable Logic and Applications (FPL'2004), Springer-Verlag LNCS 3203, pp. 1087-1089, Antwerp, Belgium, August/September 2004.

[6] Fons M., Fons F., Cantó E., López M.: "Hardware-Software Co-design of a Fingerprint Matcher on Card," Proceedings of the 2006 IEEE International Conference on Electro Information Technology (EIT'2006), Michigan, USA, May 2006

[7] Lopez M., Cantó E. and Fons M.: "Hardware-software co-design of a fingerprint image enhancement algorithm," 32nd Annual Conference of the IEEE Industrial Electronics, Paris, France, Nov. 2006.

[8] Yang S., Sakiyama K. and Verbauwhebe I.: "A Compact and Efficient Fingerprint Verification System for Secure Embedded Devices," IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2005), March 2005, pp. 609-612.

[9] Yang S., Sakiyama K. and Verbauwhebe I.: "Efficient and Secure Fingerprint Verification for Embedded Devices," EURASIP Journal on Applied Signal Processing, vol.2006, no.3, pp. 1-11, 2006

[10] Schaumont P., D. Hwang D., Verbauwhede I.: "Platform-based design for an embedded fingerprint authentication device," IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 24, no. 12, pg. 1929-1936, Dec 2005.

[11] Hung-Chih Lai, Savvides M., Tsuhan Chen, "Proposed FPGA Hardware Architecture for High Frame Rate (≫100 fps) Face Detection Using Feature Cascade Classifiers,"; Biometrics: Theory, Applications, and Systems, 2007. BTAS 2007. First IEEE International Conference on, pp. 1-6, Washington, EE.UU, Sept. 2007

[12] Ramos R., López M., Cantó E., Rodriguez L.: "SVM Speaker Verification System based on low-cost FPGA", Proceedings of the 19th International Conference on Field-Programmable Logic and Applications (FPL'2009), pp. 582-586, Prague, Czech Republic, August/September 2009.

[13] Daugman, J. G.: "How iris recognition works," IEEE Trans. Circuits Syst. Video Technology, vol. 14, no. 1, pp. 21-30, Jan. 2004.

[14] _____, "New Methods in Iris Recognition", IEEE Trans. Systems, Man, and Cybernetics – Part B: Cybernetics, vol. 37, no. 5, Oct. 2007, pp. 1167-1175.

[15] _____, "The importance of being random: Statistical principles of iris recognition," Pattern Recognition, vol. 36, 2003, pp. 279-291.

[16] Gupta R. K. and De Michelli G.: "Hardware-Software co-synthesis for Digital Systems," IEEE Design and Test of Computers, Sep. 1993, pp. 29-41.

[17] Henkel J. and Ernst R.: "An Approach to Automated Hardware/Software Partitioning using a Flexible Granularity that is Driven by High-Level Estimation Techniques," IEEE Transactions on Very Large Scale of Integration Systems, Vol. 9, No. 2, April 2001, pp. 273-289.

[18] Furber Steve: "ARM system-on-chip architecture", Second edition, Addison Wesley, 2000

[19] PIER^{TM} 2.4 Specifications, http://www.l1id.com/pages/148-specifications

[20] MicroBlaze Processor Reference Guide. Available: http://www.xilinx.com/support/

[21] Floating Point Operator v3.0, Xilinx LogicCore. Available: http://www.xilinx.com/products/ipcenter/floating_pt.htm

[22] Xilinx® Spartan™-3 Development Kit User Guide. Available: https://www.em.avnet.com

[23] Mohamed Aslam Ali: "Leonardo Spectrum Getting Started Features Document," Available: http://www.mentor.com/products/fpga_pld/synthesis/leonardo_spectrum/index.cfm

List of figure captions:

Fig. 1. Example of iris image, with graphics indicating results of localization of the inner and outer boundaries and eyelids. The bit stream in the top left results from demodulation by 2D Gabor wavelets to encode the iris pattern as a phase sequence.

Fig. 2. Illustration of non-circular boundaries for iris and pupil.

Fig 3.- General structure of a system based on hardware-software co-design.

Fig 4.- a) Algorithm for calculating the variance of an image of size NxM pixels, b) Unrolled version for hardware scheduling.

Fig 5.- Pipelined hardware scheduling

Fig 6.- Internal hardware structure of the system.

Fig 7.- Generation of signal select for multiplex management

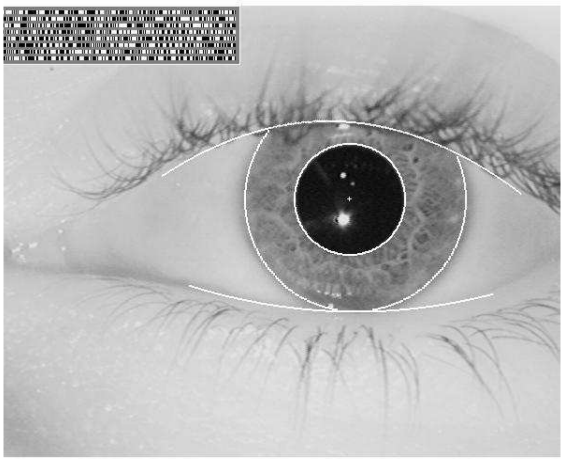


Fig. 1. Example of iris image, with graphics indicating results of localization of the inner and outer boundaries and eyelids. The bit stream in the top left results from demodulation by 2D Gabor wavelets to encode the iris pattern as a phase sequence.

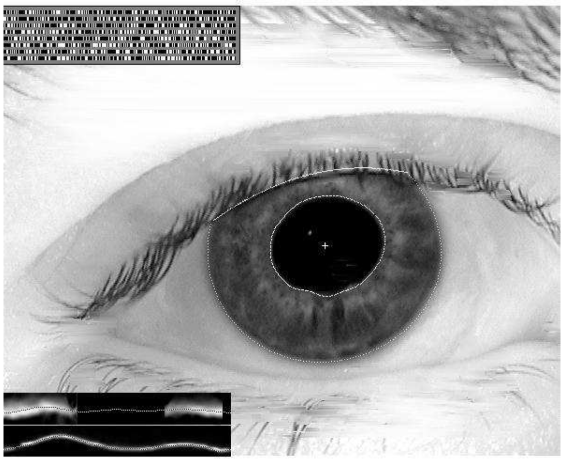


Fig. 2. Illustration of non-circular boundaries for iris and pupil.

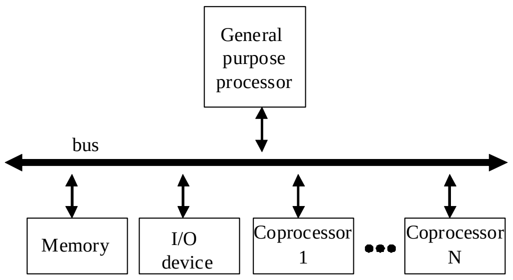


Fig 3.- General structure of a system based on hardware-software co-design.

```
// mean represents the average image intensity
// im is the image pointer

int variance (unsigned char* im, unsigned char mean) {
int i,j,irow,sum;

sum=0;
irow=0;

for (i=0; i<N; i++){
     for (j=0; j<M; j++) {
         sum+=((im [irow+j]-mean) * (im [irow+j]-mean));
     }

irow+=M;
}

return(sum/(N*M));
}
```

```
// mean represents the average image intensity
// im is the image pointer

int variance (unsigned char* im, unsigned char mean) {
int i,j,irow,sum,pixel,image,dif,pow;

sum=0; irow=0;
for (i=0; i<N; i++){
       for (j=0; j<M; j++) {      // (operation 1, group 1)
            pixel=irow+j;           // (operation 2, group 2)
            image=im [pixel];       // (operation 3, group 1 & group 2)
            dif=image-mean;         // (operation 4, group 1)
            pow=dif * dif;          // (operation 5, group 2)
            sum+=pow;               // (operation 6, group1)
        }
irow+=M;
}
return(sum/(N*M));
}
```

Fig 4.- a) Algorithm for calculating the variance of an image of size NxM pixels, b) Unrolled version for hardware scheduling.

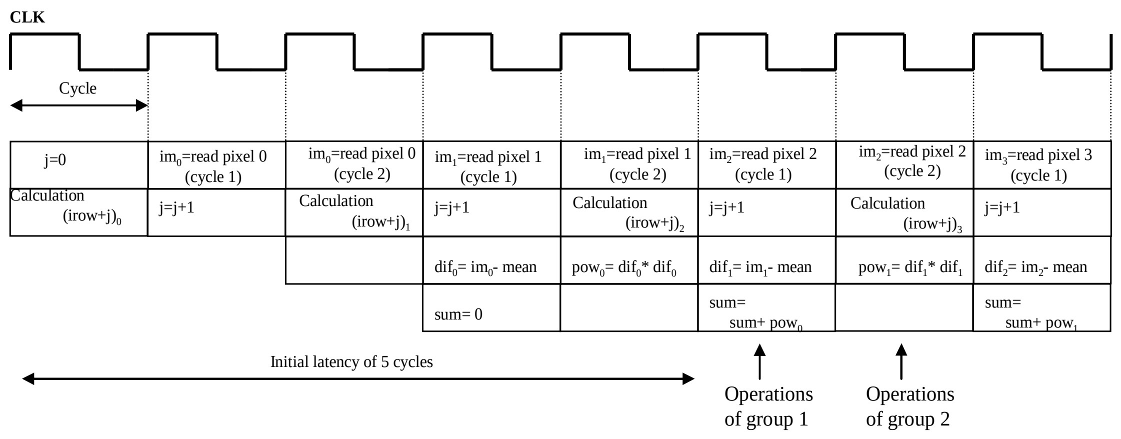


Fig 5.- Pipelined hardware scheduling.

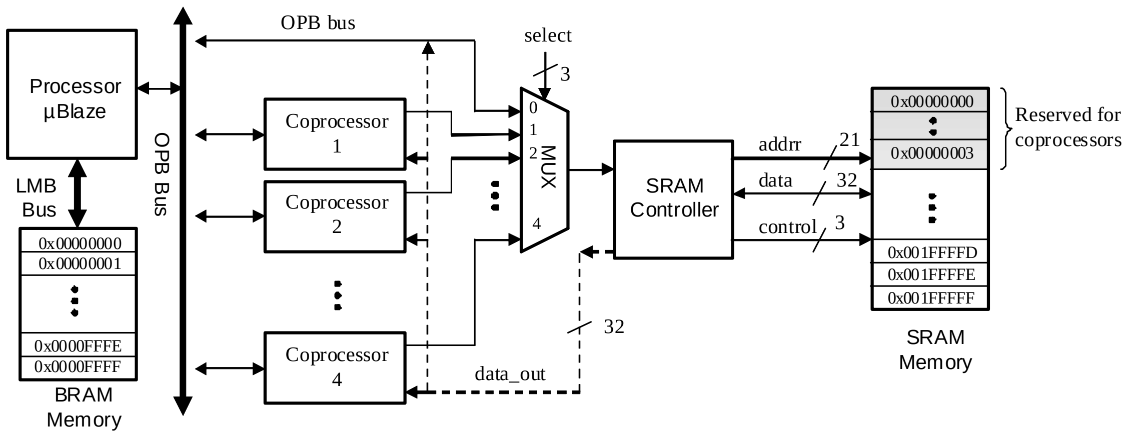
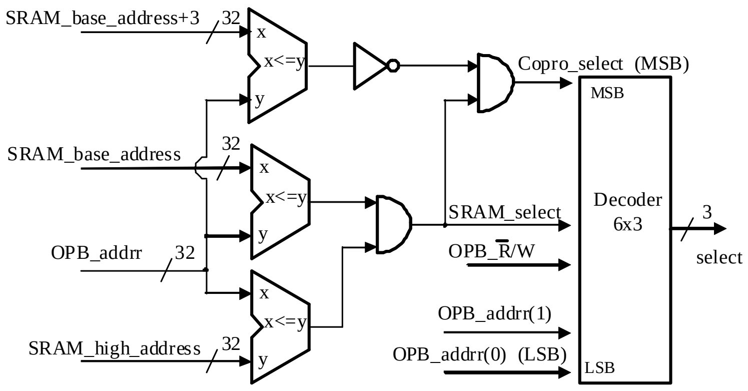


Fig 6.- Internal hardware structure of the system.



Decoder truth table

Input	select	
00xxx	xxx	(address out of range)
01xxx	000	(microprocessor)
10xxx	xxx	(not possible)
111xx	000	(microprocessor)
11000	001	(copro. 1)
11001	010	(copro. 2)
11010	011	(copro. 3)
11011	100	(copro. 4)

x means don't care

Fig 7.- Generation of signal select for multiplex management