A Vector Approach to Cryptography Implementation

Jacques Jean-Alain Fournier, Moore Simon

A Vector Approach to Cryptography Implementation

Jacques J.A. Fournier[1,2] and Simon Moore[1]

[1] University of Cambridge, Computer Laboratory, UK
{Jacques.Fournier,Simon.Moore}@cl.cam.ac.uk
[2] Gemplus S.A, La Ciotat, France
jacques.fournier@gemplus.com

Abstract. The current deployment of Digital Right Management (DRM) schemes to distribute protected contents and rights is leading the way to massive use of sophisticated embedded cryptographic applications. Embedded microprocessors have been equipped with bulky and power-consuming co-processors designed to suit particular data sizes. However, flexible cryptographic platforms are more desirable than devices dedicated to a particular cryptographic algorithm as the increasing cost of fabrication chips favors large volume production. This paper proposes a novel approach to embedded cryptography whereby we propose a vector-based general purpose machine capable of implementing a range of cryptographic algorithms. We show that vector processing ideas can be used to perform cryptography in an efficient manner which we believe is appropriate for high performance, flexible and power efficient embedded systems.

Keywords. Cryptography, AES, Montgomery Modular Multiplication, RSA, vector architecture.

1 Introduction

Given the commercial value of digital contents, their management in mobile equipments (like PDAs, mobile phones or smart-cards) has become a critical issue for content issuers. Digital Right Management (DRM) schemes are being worked on. For example the Open Mobile Alliance (OMA) is working on a DRM architecture for the mobile industry [1]. In those DRM schemes, the distribution, management and protection of data rely on the use of complex cryptographic protocols and algorithms. In such a context, the processors used (in particular those in mobile equipments) face constraints of size, power, cost, performance and security.

During the past 15 years, we saw quite a few publications about hardware modules for cryptographic applications. Most of those proposals make use of processors which are very application specific. They are not only optimized for one particular algorithm but also for particular sizes to suit market requirements. For security, counter-measures have been

proposed, most of which are software-based leading to bulkier codes and slower programs. A hardware-software co-design approach is being undertaken by other researchers [2, 3, 4] in order to have a hardware that would reduce the cost of those software counter-measures.

Our approach uses Data Parallel techniques for cryptographic applications. We first describe how we chose the vector design space. We then illustrate how cryptographic algorithms can be vectorised by giving two examples. This then takes us to the design of the corresponding vector processing machine before finally presenting results obtained on the functional simulator. With this approach, we propose an architecture which can achieve high performance and flexibility with little increase in control logic compared to scalar processors. Those characteristics of performance and flexibility are particularly relevant to DRM applications where cryptographic applications are made to run on processors having different constraints, going from the 'computer terminal' of the Rights Issuer to the small embedded chip of the DRM Agent found in a mobile equipment.

2 Having a quantitative approach

Recently, there has been an explosion in the use of cryptographic processors for embedded applications. For *secret-key* algorithms those hardware implementations can be considered to be rather straight-forward. For *Public-Key* systems however, given the complexity of the computations involved, designers have been implementing systems for static lengths (like having long-precision number multipliers for example). Some have been integrating crypto-oriented instructions into the instruction set of General Purpose Processors (GPPs) [5, 6, 7]. Others had a more scalable approach as depicted in [8]. But none have had a systematic approach where hardware designers would look for a design which would be the 'best' trade-off between speed, security, chip area and power consumption.

Having identified this need, we went back to the architecture design space and look for the best architecture that would allow us to undertake such a quantitative study. Note that this paper focuses on the micro-architecture design of a cryptographic accelerator. Issues of security (and related countermeasures) are beyond the scope of this paper.

2.1 A case for a vector architecture

According to [9], the architecture design space can be decomposed into a tree shown in Figure 1. From there, our approach was to parse through that tree and decide on the best design approach for our cryptographic algorithms.

Single Instruction Scheme Processors are chosen in order to maintain compatibility with existing smart-card chips. Having a *Multiple Instruction Scheme* would imply having a multi-processor system which does

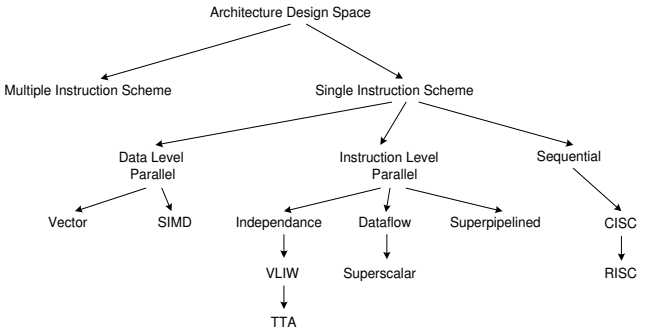

Fig. 1. Tree decomposition of the architecture design space

not fit with actual power and size constraints on embedded chips. *Instruction Level Parallel* architectures were also put aside because having parallel instruction executions:

- requires complicated instruction decoding and scheduling units, which be against our motivation of reducing complexity.
- implies the use of very sophisticated instruction decoders and issuers, which consume a lot of power as illustrated in [10]. In the latter paper, the authors that in a superscalar microprocessor where instructions are issued in parallel, the instruction issue and queue logic accounts for nearly one quarter of the total energy consumed by the processor while another quarter is accounted for by the instructions' reorder buffers.
- is not well suited for those particular applications: most cryptographic algorithms involve the sequential use of precise instructions or operations leaving little room for parallelism at this level.

A *Data Level Parallel* approach was chosen because

- the data used by those cryptographic algorithms can be decomposed into a vector of shorter data onto which operations can be applied in parallel (or *partially-parallel*) as illustrated in this paper.
- The instruction decoding is simpler, i.e. no dedicated logic is required for dynamic instructions' schedule and reordering.
- In terms of security, working on data in parallel can in theory reduce the relative contribution of each data piece to the external power consumption as announced by [11].

Hence we used Data Level Parallel techniques to design our cryptographic processing unit. Our design's vector machine is controlled by a General Purpose Processor (GPP) which also allows the optimal execution of 'scalar' codes[1].

[1] In this paper, a *scalar code* is an algorithm's code implemented on a scalar machine (MIPS-I) and a *vector code* is an algorithm's code implemented on a vector machine (VeMICry).

3 Proposed methodology for vectorizing cryptographic algorithms

We chose two case studies to illustrate how cryptographic algorithms can be vectorised: the AES symmetric key algorithm and modular multiplication based on Montgomery's algorithm (used in both RSA or Elliptic Curves Public Key Cryptography). For each of those case studies we look at their performance on a scalar MIPS-I architecture ([12, 13]). We identify the most time-consuming operations. We then show how the latter can be improved by having a vector approach based on an instruction set defined in Appendix A. In section 5 we show how these algorithms perform on our functional simulator.

3.1 Vectorising the Advanced Encryption Standard

The AES algorithm is described in [14]. The algorithm is meant to work for key lengths namely 128, 192 or 256 bits. In this study, we will concentrate on the 128-bit version of the AES as it is very representative of what's happening.

Scalar Implementation on the MIPS Our test implementation on the MIPS-IV is illustrated in Figure 2. The key schedule is done first and the sub-keys stored in RAM. The encryption process is then executed. No counter-measures are implemented. We focus on the encryption process.

Table 1 is an analysis of the time taken the different processes. This provides an indication of the most penalizing operations, in particular the `KEY-SCHEDULE`, `SUBBYTE` and `MIXCOLUMNS` operations.

Sub-Process	♯ clock-cycles	♯ times called	Total	% of total encryption
`KEY-SCHEDULE`	508	1	508	16
`ADDRNDKEY`	16	11	176	6
`SUBBYTE`	68	10	680	22
`SHIFTROWS`	26	10	260	8.5
`MIXCOLUMNS`	143	9	1287	42

Table 1. Decomposition of the AES-128 encryption

Vector Approach to AES We applied a vector approach to the encryption. We propose to vectorise the different processes as follows (based on instructions from Appendix A).

The `ADDRNDKEY` is a byte-wise XOR between the data matrix and the corresponding sub-key matrix. This operation is applied to each column (which corresponds to a 32-bit word. With our vector ISA, the `ADDRNDKEY` can be implemented in just four instructions:

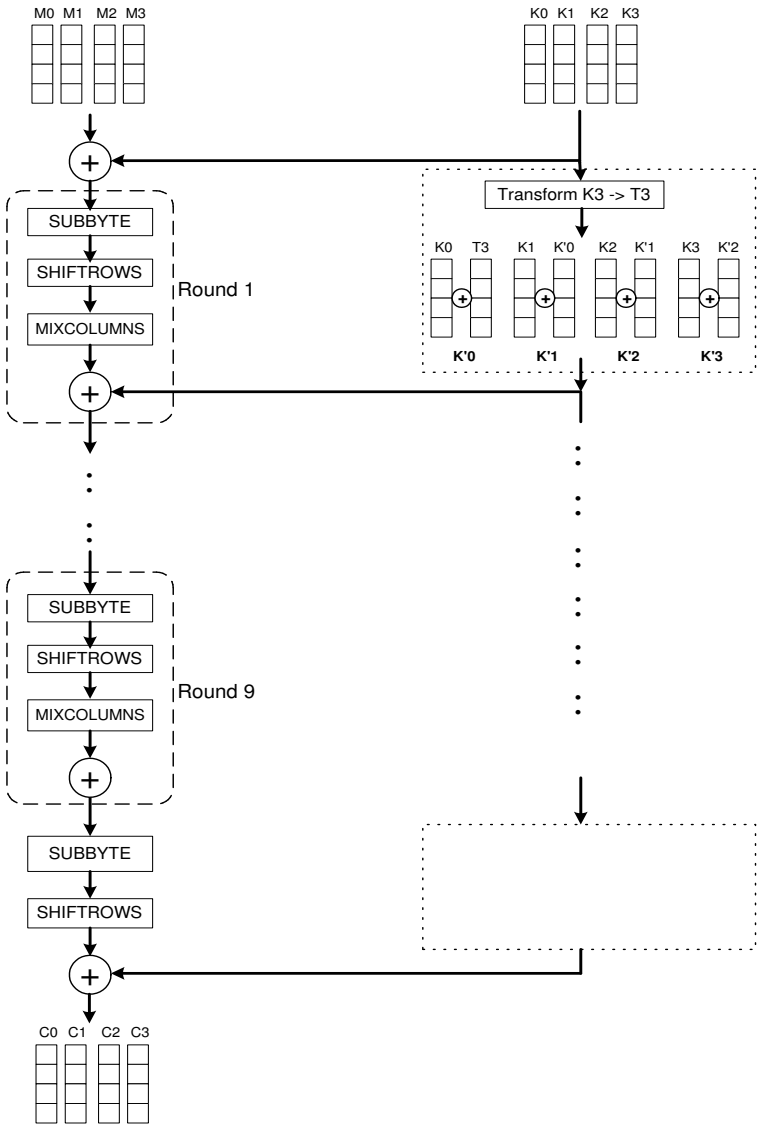

Fig. 2. AES structure

```
VLOAD   V0, (adr_key), 3    # loads 4 words into V0
                            # starting at address 'adr_key'
VLOAD   V1, (adr_data), 3
VXOR    V2, V0, V1
VSTORE  V2, (adr_data), 3
```

The SUBBYTE is a byte-wise look-up process. For this purpose we have a VBYTELD Vx, Ry, m instruction as explained in Appendix A. Such an instruction can be implemented given we have the memory organization described in Section 4.2. Note that this optimization is also useful for the KEY-SCHEDULE.

Originally the SHIFTROWS function is composed of left rotations on each row of the data matrix and if we had represented each row of the data matrix on a 32-bit word, the SHIFTROWS would have been very simple. But in our implementation, each 32-bit word is one column of the data matrix, hence the difficulty of implementing this operation. Suppose we have the operations VTRANSP and VBCROTR[2] (Vector-Bit-Conditional-Rotate-Right), the SHIFTROWS operations can be implemented as follows:

```
VLOAD   V0, (adr_data), 3   # loads 4 words of data into V0
VTRANSP V1, V0, 4           # V1 = V0 transposed
ADDIU   R11, 0x000E
MTVCR   R11                 # VCR = 1110b
VBCROTR V2, V1, 24          # V2 = V1 whose words indexed 1,
                            # 2,3 are rotated right by 24 bits
ADDIU   R11, 0x000C
MTVCR   R11                 # VCR = 1100b
VBCROTR V1, V2, 24          # V1 = V2 whose words indexed 2,3
                            # are rotated right by 24 bits
ADDIU   R11, 0x0008
MTVCR   R11                 # VCR = 1000b
VBCROTR V2, V1, 24          # V2 = V1 whose word indexed 3
                            # is rotated right by 24 bits
VMOVE   V0, V2, 4           # V0 = V2 transposed
VSTORE  (adr_data), V0, 3   # stored words indexed 0,1,2,3
                            # of V0 to address of data
```

The MIXCOLUMNS operation is the most time consuming one as shown in Table 1. It is a matrix multiplication working on each column as defined below:

$$\begin{pmatrix} a' \\ b' \\ c' \\ d' \end{pmatrix} = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \bullet \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} x \cdot a + (x+1) \cdot b + c + d \\ a + x \cdot b + (x+1) \cdot c + d \\ a + b + x \cdot c + (x+1) \cdot d \\ (x+1) \cdot a + b + c + x \cdot d \end{pmatrix} \quad (1)$$

[2] See Appendix A

such that

$$\begin{pmatrix} a' \\ b' \\ c' \\ d' \end{pmatrix} = \begin{pmatrix} x(a \oplus b) \oplus b \oplus c \oplus d \\ x(b \oplus c) \oplus a \oplus c \oplus d \\ x(c \oplus d) \oplus a \oplus b \oplus d \\ x(a \oplus d) \oplus a \oplus b \oplus c \end{pmatrix} \tag{2}$$

Each of the individual byte multiplications is done in the field GF(2^8), modulo the irreducible polynomial given by

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{3}$$

whose binary representation is `0x11B`. The central operation is hence the multiplication operation by x modulo $m(x)$. Given the instructions in Appendix A, the `MIXCOLUMNS` operation can be implemented as follows:

```
VLOAD   V0, (adr_data), 3   # loads 4 words in V0.
                            # Suppose each word of V0
                            # is made up of bytes (a,b,c,d)
ADDIU   R11, R0, 0xFFFF
MTVCR   R11                 # VCR = 0xFFFF
VBCROTR V1, V0, 8           # Each word of V1 = (d,a,b,c)
VBCROTR V2, V0, 16          # Each word of V2 = (c,d,a,b)
VBCROTR V3, V0, 24          # Each word of V3 = (b,c,d,a)
VXOR    V4, V0, V3          # Each word of V4 =
                            # (a+b,b+c,c+d,d+a)
ADDIU   R11, R0, 0x011B
MTVCR   R11                 # VCR = 0x011B
VMPMUL  V5, V4, R0          # Each byte of V4 is shifted
                            # by 1 bit left and XORed with
                            # last byte of VCR if outgoing
                            # bit is 1.
                            # Mult. by 'x' mod 0x011B.
                            # 1 word of V5 =
                            # (x(a+b),x(b+c),x(c+d),x(d+a))
VXOR    V0, V5, V1          # 1 word of V0 = (x(a+b)+d,
                            # x(b+c)+a,x(c+d)+b,x(d+a)+c)
VXOR    V0, V0, V2          # 1 word of V0 = (x(a+b)+d+c,
                            # x(b+c)+a+d,x(c+d)+b+a,x(d+a)+c+b)
VXOR    V0, V0, V3          # 1 word of V0 = (x(a+b)+d+c+b,
                            # x(b+c)+a+d+c,x(c+d)+b+a+d,
                            # x(d+a)+c+b+a)
VSTORE  V0, (adr_data), 3
```

3.2 Vectorizing Montgomery's Modular Multiplication

Two commonly used Public Key algorithms are RSA and ECC[3]. RSA is based on the modular exponentiation of large integers (typically between

[3] Elliptic Curve Cryptography

1024 to 2048 bits or more). ECC is based on the scalar multiplication of a point on an elliptic curve in a finite field (either in $\mathbb{F}_p$ with p prime or $\mathbb{F}_{2^m}$). In both cases the most critical operation is the long precision modular multiplication. In [15], the author looks at different techniques for optimally implementing the modular multiplication operation. One technique that came out of the lot, both in terms of performance and code complexity, is based on the method originally proposed by Montgomery in [16].

For our study, we looked at Elliptic Curve Cryptography over binary fields [17]. The basic modular multiplication consists of multiplying the co-ordinates of given points on the elliptic curve. Those co-ordinates have a polynomial representation and the multiplication is done modulo an irreducible polynomial in the same field. Modular multiplications have been thoroughly studied and optimized. Methods like those proposed in [18] based on Montgomery's method are quite rapid algorithms. As explained in [18], Montgomery's algorithm can be implemented to interleave the multiplication and the reduction phases. In the latter paper, the authors show that we can use Montgomery's algorithm to calculate $c(x) = a(x) \cdot b(x) \cdot r(x)^{-1} \bmod f(x)$ where $f(x)$ is an irreducible polynomial. Given that we are working in the field $\mathbb{F}_{2^m}$, the polynomials involved in this algorithm are of length m, the authors in [18] show that $r(x)$ can be chosen such that:

$$r(x) = x^k \text{ where } k = 32M \text{ and } M = \left\lceil \frac{m}{32} \right\rceil \tag{4}$$

If we suppose that the multiplicand a(x) can be decomposed into a linear combination of 32-bit polynomials denoted by $A_i(x)$ such that

$$a(x) = A_{M-1}(x).x^{32(M-1)} + A_{M-2}(x).x^{32(M-2)} + \ldots + A_0(x) \tag{5}$$

we have the algorithm in Figure 3 for a 32-bit architecture: $C_0(x)$ is the least significant 32-bit word of the polynomial $c(x)$ and $N_0(x)$ is the *'Montgomery's constant'*, which is pre-calculated, such that $N_0(x) \cdot F_0(x) \bmod x^{32} = 1$.

Input : $a(x), b(x), f(x), M$ **and** $N_0(x)$
Output : $c(x) = a(x).b(x).x^{-32M}$ **mod** $f(x)$

1. $c(x) \leftarrow 0$
2. **for** $j = 0$ **to** $M - 1$ **do**
3. $c(x) \leftarrow c(x) + A_j(x) \cdot b(x)$
4. $M(x) \leftarrow C_0(x) \cdot N_0(x)$ **mod** x^{32}
5. $c(x) \leftarrow c(x) + M(x) \cdot f(x)$
6. $c(x) \leftarrow c(x)/x^{32}$
7. **endfor**

return $c(x)$

Fig. 3. 32-bit Montgomery Modular Multiplication

Scalar implementation on MIPS On the scalar MIPS, the modular multiplication takes about 22300 clock-cycles. In this test program, we used test values from the field $\mathbb{F}_{2^{191}}$ with a modulus $f(x) = x^{191} + x^9 + 1$ allowing us to store all values in registers. We thus spare additional memory accesses.

Vector approach to modular multiplication We looked at the vector instructions that can help to enhance the execution of this 'interleaved' Montgomery Modular Multiplication. As a result of which, we obtain the following assembly code (The comments refer to the algorithm in Figure 3):

```
    .global MultBinPoly
    .ent    MultBinPoly

MultBinPoly:
    lw       $24, 16($29)          # loading data size (M)
    lw       $2, 20($29)           # loading the value of N0
    vload    $v0, $5, 5            # v0 <= b(x) on 6 words
    vload    $v1, $6, 5            # v1 <= f(x) on 6 words
    vsmove   $v3, $0, 8            # v3 cleared; (v3 == c(x))
    addiu    $15, $0, 0            # 'j': loop init
    sll      $24, $24, 2           # $24 <= 4M

LoopBin:
    add      $8, $15, $4           # add. of j-th word of a(x)
    lw       $8, 0($8)             # j-th word of a(x)
    vspmult  $v5, $v0, $8          # v5 <= a[j]*v0;(v0=b(x))
    vxor     $v3, $v5, $v3         # v3 <= v5 + v3
    vextract $9, $v3, 1            # $9 <= C_0
    vsmove   $v2, $9, 1            # v2[0] <= $9;($9=C_0)
    vspmult  $v4, $v2, $2          # v4 <= N0 * v2
    vextract $9, $v4, 1            # $9 <= M(x)
    vspmult  $v5, $v1, $9          # v5 <= v4[0]*v1;(v1=f(x))
    vxor     $v3, $v3, $v5         # v3 <= v3 + v5
    vwshr    $v3, $v3, 1           # v3 <= v3 shifted right by 1
    addi     $15, $15, 4           # Increase index by 4
                                   # as we read 4 bytes
    bne      $15, $24, LoopBin
    nop
    vstore   $7, $v3, 5
    j        $31
    nop
    .end    MultBinPoly
```

4 Proposed Architecture

Vector Processor techniques have been widely used either in super-computers like the Cray machine [19] or in Digital Signal Processing applications

like on Intel's MMX or the T0 architecture described in [20]. In the latter example, the authors already use a MIPS-like scalar processor. In this section we present the foundations for our vector architecture.

Our design aims at offering high performance for the parallel data cryptographic processes without penalizing the scalar executions. Because of this, we have an approach where we go from an already existing, highly performing, General Purpose Processor and 'plug' in the vectorial co-processor. This is particularly true with the MIPS architecture where co-processor interfaces are well defined, easing user Application Specific Extensions. The specification and definition of what we will call the Vectorial MIPS for Cryptography (VeMICry) has to be done on two levels:

- **Resource/Architectural Level:** definition of the resources present in that vectorial unit (register files, processing units, memory interface units ...).
- **Instruction scheduling and pipelining:** Specification of the vector instructions' execution with respect to the scalar pipeline in addition to the inner pipeline for each of the vector instruction.

4.1 Architectural specification

Appendix A of [21] provides a comprehensive picture of the theory behind vector processing and its application to micro-processors. To suit the MIPS 'load-store' architecture and to avoid complex memory accesses, we chose a *Register-to-Register* vector architecture: we hence hope to reduce memory-register transfers, which are the privileged attack paths for side channel analysis. Note that in this paper we work only on code implemented directly in assembly language, which means that we will not be talking about compiler optimization techniques.

4.2 Vector Register File

The structure and architecture of the vector register file will be the determinant factor in defining the rest of the architecture. 6 factors will determine the structure of our vector register file:

- m: The size of each element of the vector elements ($m = 32$).
- q: The number of such vector registers.
- p: The number of elements in each vector register. This will be called the *depth* of each vector.
- r: The number of lanes into which the vector registers are organized. This notion is borrowed from [20] where it is associated to the number of VPUs[4] available to the VeMICry. We have as many lanes as there are VPUs. Ideally we would have $r = p$ allowing us to work on the p elements in parallel: the j^{th} VPU for example would be 'associated' to a register file made of all the j^{th} elements of all the vector registers. However, in some cases, for size and power constraints we will not be allowed p VPUs. We leave r as a parameter for our analysis as to what would be the best performance to size trade-off. As

[4] Vector Processing Units

a result, the j^{th} VPU will be associated not only to the j^{th} elements across the register file but also $j + r^{th}$, $j + 2r^{th}$...

- l: The number of elements of the vector processor onto which the function is applied. Our analysis revealed that it would be interesting to work on vector lengths which are not necessarily equal to the depth of each vector register; specially in the case where $r \neq p$, both in terms of speed and power consumption. Setting the vector's length could done by setting a configuration register for example[5].
- The memory latency is also an important factor. This not only depends on the number of read and write ports per VPU but also on the definition of the interface with the memory or even how many 'memory banks' we could have in parallel. In our architecture, we propose to have a software managed memory bank *per* lane. Within each 'bank' we have 4 parallel concurrently accessible byte arrays of say 1 kilobytes each. Such a structure allows each VPU to smartly fetch four bytes in parallel, specially for the `VBYTELD` instruction.

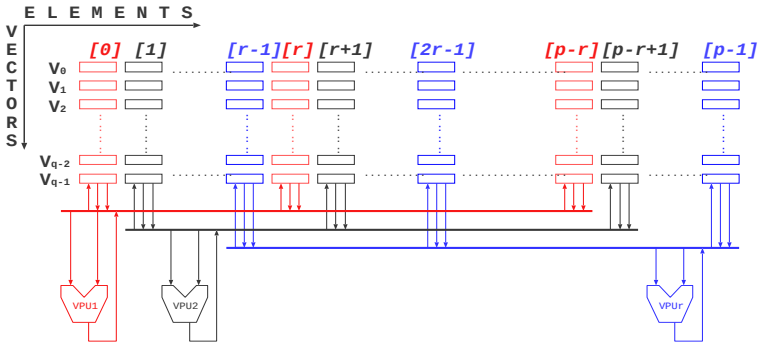

Fig. 4. Vector Register File

We obtain the register file architecture shown in Figure 4. We propose to study the influence of those 6 factors on performance and area. In addition to the Vector Registers, we identified the need for a Vector Conditional Register (VCR) which is a p bit register, a Scalar Buffer Interface (SBI) register to act as buffer from scalar values being shared between the scalar core and the vector processing unit and a CARry buffer (CAR) to store the most significant word or carry when doing addition or multiplication (in particular when $l = p$).

4.3 Vector Instruction Execution & Scheduling

In this section we briefly describe the schedule and execution of the vector instructions. A vector instruction is meant to replace what would be

[5] Note that this factor is relevant when pipeline issues come into consideration. For the functional simulator, we assume that we work on all p elements

in software a loop; a loop where the data being operated on are independent from each other and where the calculation of each iteration of the loop is independent from the calculation of the neighboring iterations. However by looking at some of the instructions in Appendix A, we can see that operations like `VADDU`, do not obey to this basic requirement. For such instructions we will take advantage of the fact that the calculation on each element of the vector is only 'partially' independent from that on its neighbors.

From then on, we define three classes of vector instructions:

Definition 1. *A `Genuinely Independent Vector Instruction` **(GIVI)** is one where the transformation applied to every element of the operand vectors is independent from the application of that same transformation on this same element's neighbors.*

Definition 2. *A `Partially Independent Vector Instruction` **(PIVI)** is one where the transformation applied to every element of the operand vectors depends partially on the result of the same operation applied to one of its neighbors.*

Definition 3. *A `Memory Accessing Vector Instruction` **(MAVI)** is a vector register-memory instruction where a memory access is required for the application of the required transformation on every element of the operand vectors.*

Each of those groups of instructions has its own dependency constraints which lead to the definition of a characteristic sequence of execution's decomposition for each group. The instruction decoding is handled by the scalar MIPS as part of its 'normal' five stage pipeline:

- **IF**: Instruction Fetch.
- **ID**: Instruction Decode.
- **EX**: (Scalar) Execution Stage.
- **DC**: Data Cache read and alignment.
- **WB**: Write Back stage.

Upon the detection of a vector instruction, each VPU enters into its own four stage pipeline:

- **Data Fetch (DF)** stage where each VPU fetches the two (depending on the instruction) elements from the target vector registers. If a scalar register is involved, the value is fetched from the latter scalar register and written back into the SBI register.
- **Execute-Multiply (EXM)** stage where the VPU performs the corresponding multiplication or addition calculation for a PIVI. For a GIVI or a MAVI, nothing is done.
- **Execute-Carry (EXC)** stage where the 'carry' selection is done for the PIVIs and the latter's calculation is completed. For a GIVI or a MAVI, the corresponding calculation/manipulation is done onto the arguments fetched in stage DF.
- **Write Back (WB)** stage where the result from the VPU is written back to the corresponding element of the destination vector register.

It is left to the software to make sure the vector register length is properly set before doing any vector instruction when working on vectors of length $l < p$.

GIVI execution Let's consider the general case where p is 'too' large and that we only have r VPUs where $r \leq p$ (could be specially true for embedded processors). This means each VPU will have to enter $\frac{p}{r}$ times in order to apply the required operation on all p elements of the targeted vector registers as shown in Figure 5. Hence the next vector instruction will only be issued $\frac{p}{r}$ cycles later.

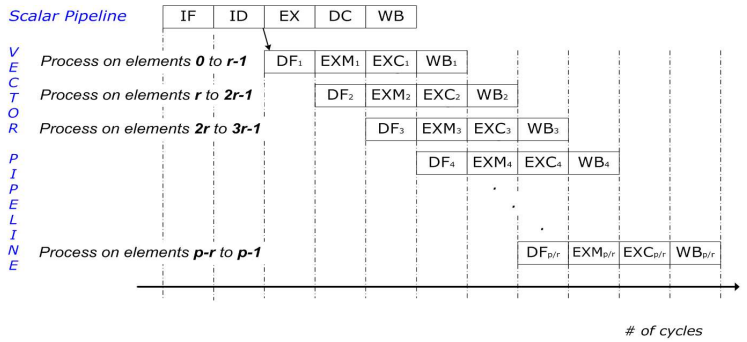

Fig. 5. Timing relationship between scalar & vector executions

PIVI execution In a *Partially Independent Vector Instruction*, the calculation on every element of the vector register depends on the calculation of the neighboring elements: the functions concerned by this category are `VADDU`, `VSPMULT`, `VSAMULT` and `VTRANSP`. For the optimal schedule of the PIVI instructions we will assume that each VPU has an internal 'temporary' 32-bit register. Most the above mentioned instructions have to handle the addition of vector elements and to anticipate on the carry being propagated from the neighboring least significant element. To do so, we assume that each VPU has a 32-bit Carry Select Adder (CSA): at each addition step the addition is performed for both cases where 'incoming' carry is 0 or 1 and the 'correct' output is determined once the correct carry is known. Like this the PIVI instruction can be made to have the same instruction issue rate as the GIVI.

MAVI execution Looking back at the Appendix, we have three MAVI instructions: `VBYTELD`, `VLOAD` and `VSTORE`. Each VPU has its own software managed memory which is the VPU can access by bytes (with 4 bytes in parallel) for the `VBYTELD` instruction and by 32-bit words for `VLOAD` and `VSTORE`. With such an arrangement the issue rate would be $\frac{p}{r}$.

Vector instructions' chaining and hazards If we work on vector depths which are greater than the number of VPUs, an instruction may take several iterations as illustrated Figure 5 for a GIVI instruction. The main type of hazard we might be confronted which is data hazard. Data hazards occur when the instruction I has as operand the result from the preceding instruction $I-1$. With our vector operations, data hazards occur when an instruction takes only 1 or 2 iterations (i.e. $\frac{p}{r} \leq 2$). For instructions having a larger number of iterations, the latency incurred by the multi-iteration process diffuses the data dependency. The following table describes the different data hazards that might occur between an instruction $I-1$ and the instruction I and how, when this is possible, pipeline stalls can be avoided by using data feed-forward mechanisms.

I-1	I	Description	Stall?	Bypass Required
GIVI	GIVI	Calculation done at EXC stage	No stall	Data forwarded from the EXC stage of $I-1$ to the EXC stage of I
GIVI	PIVI	PIVI needs result at EXM stage	Pipeline stalls after ID stage	Data forwarded from the EXC stage of $I-1$ to the EXM stage of I
PIVI	GIVI	PIVI needs result at the EXM stage	No stall	Data forwarded from the EXC stage of $I-1$ to the EXC stage of I
PIVI	PIVI	PIVI needs result at the EXM stage	Pipeline stalls after ID stage	Data forwarded from the EXC stage of $I-1$ to the EXM stage of I

Table 2. Data Dependencies on the vector instructions

5 Functional Simulation

We started by building a functional simulator for our VeMICry architecture: a functional architecture allows us to test the vector code presented in Sections 3.1 and 3.2. Moreover, with such a simulator, we can perform performance studies in terms of instruction cycles and see the effect of the parameters from Section 4.2.

5.1 Use of the ArchC simulation tool

The ArchC tool is an architecture description language which is developed by the Computer Systems Laboratory of the Institute of Computing of the University of Campinas (`www.archc.org`). The tool allows to build an architectural instruction simulator which is composed on:

– A language description used to describe the target architecture including the memory hierarchy (**AC_ARCH**) and the instruction set architecture (**AC_ISA**).
– A simulator generator (**ACSIM**) which uses the above description language to generate a Makefile which is then used for building a **SystemC** model.

It is based on a widely used commercial tool like `SystemC` [22] and allows to build quite simple architectures which is sufficient for our immediate needs. Moreover, the simulation software builder is based on GCC (`www.gnu.org`). Hence it is easy to modify the instruction set. The idea behind this study is to build a simulator of our VeMICry architecture to test the vector instructions described in A and perform some preliminary performance studies in terms of instruction cycles.

5.2 Building the functional model

Our architecture is based on the 32-bit MIPS architecture with an instruction set fully compatible with the (basic) MIPS-I family. Moreover, we hacked GCC's Assembler to compile our vector codes.

The backbone of the VeMICry model is composed of the definition files of the `MIPS-I` model which we have upgraded to add our vector instructions. In our model model:

– We have 8 vector registers ($q = 8$).
– Each vector is composed of $8 \times$ 32-bit elements ($p = 8$).
– We have 8 VPUs working in parallel ($r = 8$). Hence there are eight lanes where in the j^{th} lanes the j^{th} VPU works across the j^{th} elements of the vector registers.
– We assume that each instruction is executed in 1 cycle (only a functional model).

The simulator generates a series of basic statistics like the sequence of instructions executed (`vemicry.dasm`), a trace of the Program Counter (`vemicry.trace`) and the occurrences of each instruction along with the number of cycle-counts (`vemicry.stats`).

5.3 Functional simulation of vectorised AES

As explained previously, the vector instructions are used to optimize the `SHIFTROWS`, `MIXCOLUMNS`, `ADDROUNDKEY` and `SUBBYTE` operations. The `KEY_SCHEDULE` is implemented as a separate routine.

We validated the results generated by our vector AES encryption code. Simulations show that encrypting 16 bytes (for an AES-128) takes 160 instruction cycles. In addition to this the `KEY_SCHEDULE` took 246 instruction cycles. Those figures represent a large gain in performance when compared to the same algorithms implemented the scalar MIPS. For the scalar code the key schedule took 519 instruction cycles and the encryption took 3283 cycles.

More performance gain is achieved when we encrypt larger data files. We ran simulations where we encrypted 32 bytes with one same key, i.e. we ran the `KEY_SCHEDULE` once and the encryption codes was modified to work on 8 words of each vector register. Encrypting 32 bytes took 182 instruction cycles. This illustrates a major advantage of our architecture: depending on the depth of vector registers, we are able to encrypt large data tables with little performance penalties.

Another big advantage with our approach is that robust software countermeasures (like those described in [2]) can be implemented to compensate for any side-channel information leakage.

5.4 Simulation of vectorised Montgomery multiplication in binary fields

On the VeMICry, the calculation of the Montgomery's constant is executed in 22 instruction cycles. The main part of the modular multiplication takes 97 instruction cycles. The same modular multiplication operation takes 22331 instruction cycles on the scalar MIPS.

Note that our test values are taken from the field $GF(2^{191})$, which means that the data values have a maximum length of 192 bits. Given that in the actual architecture each vector register has 8 elements, each vector register is used to hold the 192 bits of each variable. With a depth of 8, we could work on up to 256-bits ECC (with the same number of instruction cycles), which would be far from what would be required for the next 20 years or so.

Note that in the preceding example, we perform a reduction by 32 bits each time. However, one could envisage to perform a reduction by 64 bits as this would mean that we would have half as many loops. In the algorithm depicted in Figure 3, each word is on 64 bits, which means that the calculated N_0 is also on 64 bits and also the we shift by 64 bits in the end. We only perform half the number of loops.

We modified the vector code presented at the end of section 3.2 to emulate this reduction by 64 bits. The calculation of N_0 took 72 instruction cycles and the modular multiplication itself took 84 instruction cycles. Note that N_0 can be calculated only once at the beginning of the signature algorithm and hence for comparing performances, we focus only on the multiplication algorithm. Performance gain when doing a 64-bit reduction is of the order of 13% compared to the same algorithm implemented with a reduction by 32 bits. This gain is achieved at the expense of one additional vector register.

6 Ongoing research

To have a significant quantitative study, it makes sense to study modular multiplications on larger values like in RSA. So the next phase of the

study is to test the modular multiplication on 1024 to 2048 bit values and see how the number of instruction cycles changes by varying the different sizes of the vector architecture. Then we will be implementing a synthesisable Verilog model to add the 'gate count' parameter to our benchmark.

7 Conclusion

In this paper we proposed a vector architecture for embedded cryptography. We have shown how the vector approach is relevant to cryptography and how cryptographic algorithms can be efficiently vectorised. We built and validated a functional model of our vector architecture. The vector architecture combined with our proposed instructions have helped us to reduce the number of cycles taken for an AES encryption from 3283 on the MIPS-I to 160 on the VeMICry. Likewise, modular multiplication in the field $GF(2^{191})$ has been reduced from 22331 instruction cycles to 84 cycles. We can anticipate that each lane will be at least (if not less) complex than a scalar MIPS. This would mean that our vector approach is a sound one given the performance figures measured. Further research is currently being done to study the complexity of our vector architecture and find the best trade-off between performance, size and power consumption.

References

[1] Open Mobile Alliance, "DRM Specification V2.0 Candidate Version 2.0 - 26 April 2005," Tech. Rep. OMA-DRM-DRM-V2_0-20050426-C, Open Mobile Alliance (OMA), April 2005.

[2] M.-L. Akkar and C. Giraud, "An Implementation of DES and AES, Secure against Some Attacks," *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systmes (CHES 2001)*, vol. LNCS 2162, pp. 309–318, 2001.

[3] J. Zambreno, A. Choudhary, R. Simha, and B. Narari, "Flexible Software Protection Using Hardware/Software Codesign Techniques," *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'04)*, vol. 01, no. 1, p. 10636, 2004.

[4] J.-F. Dhem and N. Feyt, "Hardware and Software Symbiosis helps smart card evolution," *IEEE Micro*, vol. 21, no. 6, pp. 14–25, 2001.

[5] J. P. McGregor and R. Lee, "Architectural techniques for accelerating subword permutations with repetitions," *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, vol. 11, pp. 325–335, June 2003.

[6] J. Groszschaedl and G. Kamendje, "Instruction Set Extension for Fast Elliptic Curve Cryptography over Binary Finite Fields GF(2^m)," *Proceedings of IEEE International Conference on Application Specific Systems Architectures and Processors (ASAP2003)*, pp. 455–468, June 2003.

[7] MIPS-Technologies, "SmartMIPS ASE," `http://www.mips.com/content/Products/`.

[8] A. F. Tenca and Çetin K. Koç, "A Scalable Architecture for Montgomery Multiplication," *Proceedings of CHES'99*, vol. LNCS, no. 1717, pp. 94–108, 1999.

[9] H. Corporaal, *MicroProcessor Architectures : from VLIW to TTA*.

[10] D. Folegnani and A. González, "Energy Effective Issue Logic," *Proceedings of* 28^{th} *Annual International Symposium on Computer Architecture 2001 (ISCA'2001)*, pp. 230–239, June-July 2001.

[11] E. Brier, C. Clavier, and F. Olivier, "Optimal Statistical Power Analysis," *Cryptology ePrint Archive,* `http://eprint.iacr.org/`, vol. Report 2003, no. 152, 2003.

[12] "MIPS*pro*^{TM} Assembly Language – Programmer's Guide," Tech. Rep. 007-2418-001, Silicon Graphics Inc., 1996.

[13] "MIPS^{TM} Architecture For Programmers Volume II: The MIPS32^{TM} Instruction Set," Tech. Rep. MD00086, Revision 0.95, MIPS Technologies, 1225 Charleston Road, Mountain View, CA 94043-1353, March 2001.

[14] NIST, "Specification for the Advanced Encryption Standard," Tech. Rep. 197, Federal Information Processing Standards, November 26 2001.

[15] J.-F. Dhem, *Design of an efficient public-key cryptographic library for* RISC*-based smart-cards*. PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, May 1998.

[16] P. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, vol. 44, pp. 519–521, April 1985.

[17] I. Blake, G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography*, vol. 265 of *Lecture Note Series*. London Mathematical Society.

[18] C. Koç and T. Acar, "Montgomery Multiplication in GF(2^m)," *Designs, Codes and Cryptography*, vol. 14, pp. 57–69, 1998.

[19] R. M. Russell, "The CRAY-1 Computer System," *Communications of the ACM*, vol. 21, pp. 63–72, January 1978.

[20] K. Asanovìc, *Vector Microprocessors*. PhD thesis, University of California, Berkeley, Spring 1998.

[21] J. L. Hennessy and D. A. Patterson, *Computer* A*rchitecture:* A Q*uantitative* A*pproach*. Morgan Kaufmann, 3 ed., 2003.

[22] T. A. Team, "The Archc Architecture Description Language - Reference Manual," Tech. Rep. v1.2, University of Campinas, `http://www.archc.org/, December 2004.`

A Vector Instructions

The VeMICry processor is composed of two families of instructions: the *scalar* instructions which correspond to the conventional MIPS-I instruction set and the *vector* instructions tailored to suit cryptographic requirements.

Suppose we have a vector processor having q vector registers. Each vector register is a vector of p words of 32 bits each. We also have a Vector Condition register (VCR) which contains p bits and which is used for conditional vector instructions to show if the condition is applied to each of the individual words of the vector. Moreover, we have a second 'scalar' register called the Carry Register (CAR) which, for some instructions, 'carry bits/words' are written back. We also assume that we are able to work on an arbitrary vector length l with, of course, $l \leq p$.

	V_i	corresponds to the i^{th} vector register
	R_j	corresponds to the j^{th} scalar regsiter
	n	16-bit immediate value
VADDU	V_l, V_j, V_k	performs the unsigned addition between the i^{th} elements of V_j and V_k, writing the result as the i^{th} element of V_l. The carry is propagated and added to the $i + 1^{st}$ element of V_l. The carry from the addition of the corresponding p^{th} words is added to the content of CAR if $l = p$.
VBYTELD	V_l, R_i, n	each word of V_l is treated as four bytes. Each byte is an offset which is added to the address stored in R_i and the byte stored at that address is read from the VPU's corresponding memory. The read byte is written to the same location as that of its original corresponding byte. This process is executed for n words of V_l.
VLOAD	V_l, R_i, n	loads in V_l the n consecutive 32-bit words from memory starting from address stored in R_i with a stride of 1 (*The notion of stride is introduced in Annexe A of [21]. A stride of '1' means that the words that are consecutively stored in the vector register are fetched by parsing the specified memory with a step of 1 word unit*) .

VBCROTR	V_l, V_j, n	The Vector-Bit-Conditional-Rotate-Right operates on each i^{th} word of V_j. If VCR[i] is 1, then $V_j[i]$ is rotated by n bits to the right and the result is written to $V_l[i]$. If VCR[i] is 0, then $V_j[i]$ copied to $V_l[i]$ without transformation
VEXTRACT	R_i, V_j, n	copies the value of the $V_j[n-1]$ into R_i. If $n = 0$, then it is CAR which is written to scalar register
VTRANSP	V_l, V_j, n	copies vector in V_j to register V_l. If n is zero, there is a direct copy without transposition. If n is non-zero, V_j is viewed as a $4 \times p$ matrix which is transposed and written to vector register V_l with a stride of n
VMPMUL	V_l, V_j	The Vector Modular Polynomial Multiplication treats each i^{th} word of V_j as four bytes: each byte is a polynomial in $GF(2^8)$ which is multiplied by x modular the polynomial represented in the 9 least significant bits in scalar register VCR. The result is written to V_l
VSADDU	V_l, V_j, R_k	Vector-Scalar-Addition does the unsigned arithmetic addition of value in R_k to every i^{th} word of V_j and writes the result to V_l. The carry is not propagated but is instead written as the i^{th} bit of the register CAR
VSAMULT	V_l, V_j, R_k	Vector-Scalar-Arithmetic-Multiplication: multiplies R_k by $V_j[p]\|\|V_j[p-1]\|\| \ldots \|\|V_j[0]$ with carry propagation and result is written to V_l. The most significant carry bits are written to register CAR
VSMOVE	V_l, R_k, n	copies the value in register R_k to the first n words of V_l. If n is *zero*, then R_k is copied to every word of V_l

VSTORE	R_k, V_l, n	stores the first n consecutive 32-bit words from register V_l to memory starting from address stored in R_k with a stride of 1
VSPMULT	V_l, V_j, R_k	Vector-Scalar-Polynomial-Multiplication: does the polynomial multiplication of R_k by $V_j[p]\|\|V_j[p-1]\|\|\ldots\|\|V_j[0]$ and the result is written to V_l. The most significant $p+1^{st}$ word is written to the register CAR
VXOR	V_l, V_j, V_k	XORs corresponding words between V_j and V_k and stores the result in V_l
VWSHL	V_l, V_j, n	Vector-Word-Shift-Left shifts the contents of vector V_j by n positions to the left inserting *zeros* to the right. The resulting vector is written to V_l and the outgoing word to CAR
VWSHR	V_l, V_j, n	Vector-Word-Shift-Right shifts the contents of vector V_j by n word position to the right inserting the data stored in CAR to the left. The resulting vector is written to V_l.
MTVCR	R_j	Writes to VCR the value contained in the scalar register R_j.
MFVCR	R_j	Copies the value contained in VCR to the scalar register R_j.