Neural, Parallel, and Scientific Computations 22 (2014) 609-622

PROCESSING ENCRYPTED SCALAR/VECTOR DATA ON UNIFIED DATAPATH

MOSTAFA I. SOLIMAN[1, 2]

[1] Computer Science and Information Department, Community College,
Taibah University, Al-Madinah Al-Munawwarah 2898, Saudi Arabia.

[2] Computer and System Section, Electrical Engineering Department, Faculty of Engineering,
Aswan University, Aswan 81542, Egypt.

mossol@ieee.org and mossol@yahoo.com

Abstract. This paper extends our proposed processor VVSHP by encryption/decryption/key-expansion units based on RC5 cryptographic algorithm to process encrypted scalar/vector data. VVSHP has a modified five-stage pipeline for executing multi-scalar/vector instructions by fetching 128-bit VLIW instruction, decoding/reading operands of four individual instructions, executing four scalar/vector operations, accessing memory to load/store 128-bit data, and writing back up to four 32-bit results. The key-expansion unit accepts 96-bit user's secrete key to generate the expanded key array needed for the encryption and decryption units. By extending the execute stage with encryption unit and memory access stage by decryption unit, CryptoVVSHP can process encrypted 32-bit data with lengths varying from 1 to 256. Thus, before storing into memory, scalar/vector data are encrypted, and after loading from memory, scalar/vector data are decrypted. Therefore, data only ever exists as plaintext inside the processor itself. The design of the proposed CryptoVVSHP processor is implemented using VHDL targeting the Xilinx FPGA Virtex-5, XC5VLX110T-3FF1136 device. The number of LUT flip-flop pairs used for implementing CryptoVVSHP is 109737, where the numbers of unused flip-flops, unused LUTs, and fully used LUT flip-flop pairs are 65178, 40904, and 3655, respectively. The complexity of CryptoVVSHP is about 23% higher than VVSHP.

Keywords - Encryption/decryption; RC5; vector processing; VLIW; unified datapath; FPGA/VHDL implementation.

1. INTRODUCTION

Nowadays, many applications need security and privacy of data. Encryption techniques are used for translating plain text data (plaintext) into something that appears to be random and meaningless (ciphertext). Conversely, decryption is the process of

Received May 19, 2014

converting ciphertext back to plaintext. In contrast to public-key cryptography, a symmetric key algorithm like RC5 [1] uses the same key for both encrypting plaintext and decrypting ciphertext. Normally, encrypted data needs to be decrypted before processing. However, hardware modifications can be made to a computer allowing the data to exist in decrypted form inside its CPU, but such that the decrypted data is not externally accessible [2]. Sagedy [3] proposed encrypted MIPS processor that can execute encrypted instructions and can operate on encrypted scalar data. Structural modifications necessary to execute encrypted instructions were identified. Moreover, cryptographic modules based on the DES algorithm were incorporated into MIPS pipeline to decrypt instructions, encrypt data being written to memory, and decrypt data being read from memory. Ahituv et al. [4] presented three advantages for encrypted processing: (1) strengthening of data security, (2) considerable savings in computer time, and (3) savings in the costs of handling part of the security problems of the operating system. They developed an algorithm to prove the feasibility of the processing data that are in an encrypted mode. Thus, one can perform arithmetic operations on encrypted data without the need to convert the data back to its non-encrypted origin before performing the required operations. On the other hand, many algorithms based on homomorphic functions have been proposed for processing encrypted data [5, 6]. Such functions enable the processing of the operation in an encrypted way and even the decryption of the result, but are lacking in terms of the encryption strength [4].

This paper follows the direction of modifying the processor architecture to allow the data to exist in decrypted form internally; however, decrypted data is not externally accessible. The proposed processor architecture processes not only encrypted scalar but also encrypted vector data. The main idea behind our proposed processor is that load/store instructions will automatically cause the appropriate decryption/encryption operations to be performed on scalar/vector data. Thus, after loading encrypted data from memory, they are decrypted and then stored in the register file. Conversely, before storing data from register file to memory, they are encrypted.

This paper extends our proposed processor VVSHP [7] by encryption/decryption/key-expansion units based on RC5 encryption algorithm to process encrypted scalar/vector data. VVSHP merges VLIW and vector processing techniques for a simple, high-performance processor architecture. On unified parallel datapaths, VVSHP processes multiple scalar instructions packed in VLIW and vector instructions by issuing up to four scalar/vector operations in each cycle. However, it cannot issue more than one memory operation at a time, which loads/stores 128-bit scalar/vector data from/to data memory. Four 32-bit results can be written back into VVSHP register file per clock cycle.

The CryptoVVSHP presented in this paper loads/decrypts or encrypts/stores 128-bit scalar/vector data from/to data memory. The encryption and decryption are based on the well-know RC5 algorithm [1]. By extending the execute stage with encryption unit and memory access stage by decryption unit, CryptoVVSHP can process 32-bit encrypted data with lengths varying from 1 (scalar) to 256 (vector). The design of our proposed

CryptoVVSHP processor is implemented using VHDL targeting the Xilinx FPGA Virtex5, XC5VLX110T-3FF1136 device [8].

The rest of this paper is organized as follows. Section II describes in detail RC5 since the extended encryption/decryption/key-expansion units to VVSHP are based on RC5 cryptographic algorithm. In addition, it presents the FPGA implementation of encryption/decryption/key-expansion units. The architecture of our proposed CryptoVVSHP processor is described in details in Section III. Section IV presents the FPGA implementation of the CryptoVVSHP on Xilinx Virtex-5. Finally, Section V concludes this paper and gives directions for future work.

2. FPGA IMPLEMENTATION OF RC5 ALGORITHM

RC5 is designed by Rivest, which is a symmetric-key block cipher notable for its simplicity [1]. A key feature of RC5 is the heavy use of data-dependent rotations. RC5 has a variable word size, a variable number of rounds, and a variable length secret key. For encrypting/decrypting scalar/vector data on CryptoVVSHP, this section presents the FPGA implementation of the encryption, decryption, and key-expansion units based on RC5 algorithm with the following parameters:

- the number of rounds (r) equals 8,
- the size of the expanded key table ($t = 2*(r+1)$) equals 18,
- the word size in bits (w) equals 16,
- the word size in bytes ($u = w/8$) equals 2,
- the number of bytes in the secrete key (b) equals 12,
- the number of words in the secrete key ($c = \lceil b/u \rceil$) equals 6,
- the number of iterations of the key-expansion module ($n = 3*\max(t, c)$) equals 54,
- the constant P_{16} used in the key-expansion module equals $(b7e1)_{16}$ or $(1011011111100001)_2$, where $P_w = \text{Odd}((e-2)*2^w$ and $e = 2.718281828459$, and
- the constant Q_{16} used in the key-expansion module equals $(9e37)_{16}$ or $(1001111000110111)_2$, where $Q_w = \text{Odd}((\phi-1)*2^w$ and $\phi = 1.618033988749$.

2.1 RC5 Encryption: Description and FPGA Implementation

The encryption module of RC5 accepts a block of data in two w-bit inputs A_0 and B_0, as shown in Figure 1. Moreover, it accepts the expanded key array $S[0:t-1]$, which stores the round keys generated by the key-expansion module. After r rounds, the encryption module generates an encrypted block in two w-bit outputs A_r and B_r. Listing 1 presents the pseudo-code of the RC5 encryption algorithm, where the main operations are addition (+), XOR ($\oplus$) and shift-left (<<<).

Listing 1: RC5 encryption algorithm

$A_0 = A_0 + S[0]$
$B_0 = B_0 + S[1]$
for $i = 1$ to r
$A_i = ((A_{i-1} \oplus B_{i-1}) <<< B_{i-1}) + S[2*i]$
$B_i = ((B_{i-1} \oplus A_i) <<< A_i) + S[2*i+1]$

Figure 1 shows the block diagram of the unrolled encryption module of RC5 algorithm, where a single clock cycle is required for encrypting $2\times w$-bit. It is clear that $(2r+2)\times w$-bit adders, $2r\times w$-bit shift-left, and $2r\times w$-bit XOR are needed for implementing the encryption module with unrolling r rounds. Figure 2 shows the number of LUTs and operating frequencies of the RC5 encryption as increasing the number of unrolled rounds from 5 to 15. To compromise between hardware complexity, frequency, and security, the

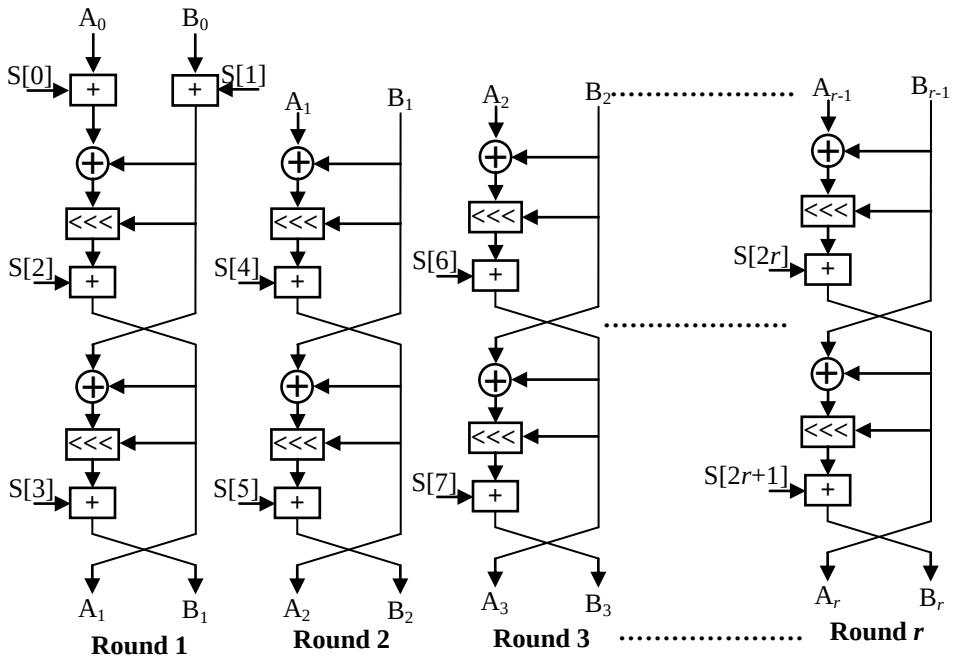


Fig. 1. Unrolled RC5 encryption algorithm with r rounds.

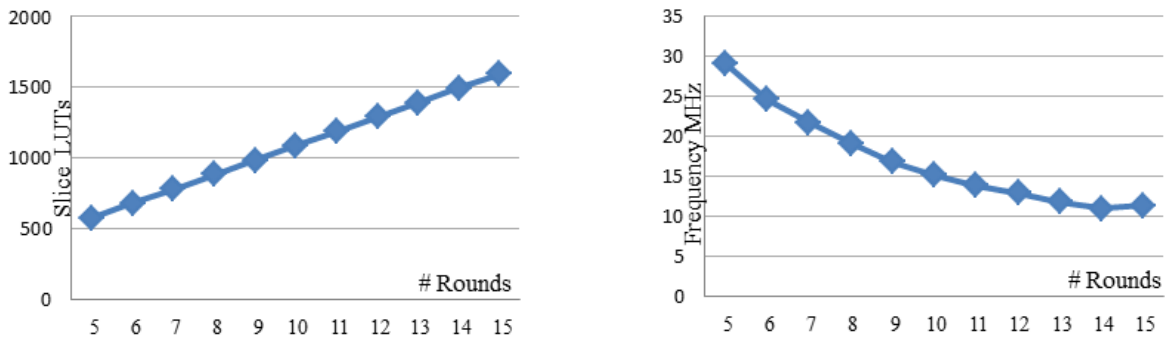


Fig. 2. FPGA implementation of the RC5 encryption algorithm with r rounds.

Table 1: HDL synthesis report of the RC5 encryption unit

Macro Statistics	16-bit adder: 18 1-bit register: 1 32-bit register: 1 16-bit xor2: 16
Slice Logic Utilization	Number of Slice LUTs: 954 Number used as Logic: 954
Slice Logic Distribution	Number of LUT Flip Flop pairs used: 954 Number with an unused Flip Flop: 954 Number with an unused LUT: 0 Number of fully used LUT-FF pairs: 0
IO Utilization	Number of IOs: 356 Number of bonded IOBs: 356 IOB Flip Flops/Latches: 33
Timing (ns) Summary	Minimum period: No path found Minimum input arrival time before clock: 54.863 Maximum output required time after clock: 2.775 Maximum combinational path delay: No path found

number of rounds (r) is selected to be eight. Moreover, Table 1 presents the statistics of hardware implementation of the encryption unit with unrolling eight rounds on Virtex-5 XC5VLX110T. The number of LUT flip-flop pairs used is 954, where the numbers with unused flip-flops, unused LUTs, and fully used LUT-FF pairs are 954, 0, and 0, respectively.

2.2 RC5 Decryption: Description and FPGA Implementation

By reversing the operations, the decryption process can be easily derived from the encryption algorithm. Listing 2 presents the pseudo-code of the RC5 decryption algorithm, where the main operations are subtraction (–), XOR (⊕) and shift-right (>>>).

Listing 2: RC5 decryption algorithm

for $i = r$ downto 1

$B_{i-1} = ((B_i - S[2*i+1] >>> A_i) \oplus A_i$

$A_{i-1} = ((A_i - S[2*i] >>> B_{i-1}) \oplus B_{i-1}$

$B_0 = B_0 - S[1]$

$A_0 = A_0 - S[0]$

Like encryption, the unrolled decryption of RC5 algorithm is implemented to decrypt $2\times w$-bit in a single clock cycle. $(2r+2)\times w$-bit subtractors, $2r\times w$-bit shift-right, and $2r\times w$-bit XOR are needed for implementing the decryption module with unrolling r rounds. Figure 3 shows the number of LUTs and operating frequency of the RC5 decryption module as increasing the number of unrolled rounds from 5 to 15. Like encryption module, the number of rounds is selected to be eight to compromise between hardware complexity, frequency, and security. In addition, Table 2 presents the statistics of hardware implementation of the decryption unit with unrolling eight rounds on Virtex-5 XC5VLX110T. The number of LUT flip-flop pairs used is 1062, where the numbers with

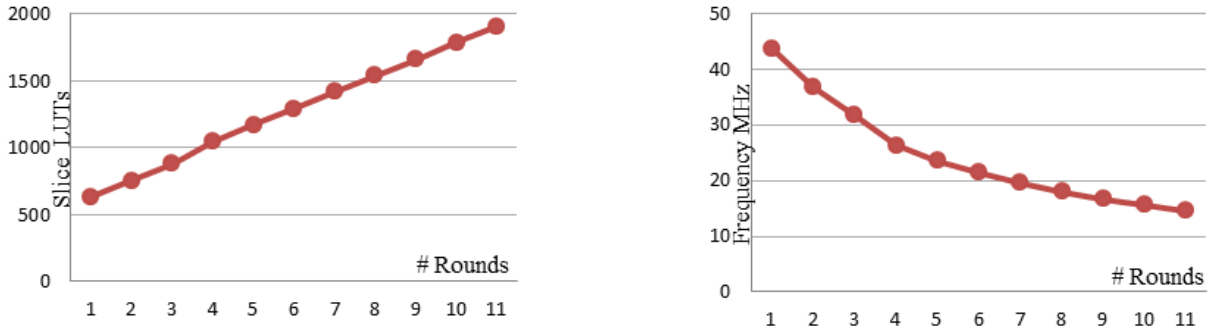


Fig. 3. FPGA implementation of decryption algorithm with *r* rounds.

Table 2: HDL synthesis report of the RC5 decryption unit

Macro Statistics	16-bit subtractor: 18 1-bit register: 1 32-bit register: 1 16-bit xor2: 16
Slice Logic Utilization	Number of Slice LUTs: 1062 Number used as Logic: 1062
Slice Logic Distribution	Number of LUT Flip Flop pairs used: 1062 Number with an unused Flip Flop: 1062 Number with an unused LUT: 0 Number of fully used LUT-FF pairs: 0
IO Utilization	Number of IOs: 356 Number of bonded IOBs: 356 IOB Flip Flops/Latches: 33
Timing (ns) Summary	Minimum period: No path found Minimum input arrival time before clock: 38.704 Maximum output required time after clock: 2.775 Maximum combinational path delay: No path found

unused flip-flops, unused LUTs, and fully used LUT-FF pairs are 1062, 0, and 0, respectively.

2.3 RC5 Key-Expansion: Description and FPGA Implementation

The key-expansion module expands the user's secret key K to fill the expanded key array S, where S resembles an array of $t = 2*(r+1)$ random binary words determined by K. The key-expansion algorithm uses two "magic constants": $P_w = \text{Odd}((e-2)*2^w$ and $Q_w = \text{Odd}((\phi-1)*2^w$, where e is the base of natural logarithms (2.718281828459), ϕ is the golden ratio (1.618033988749), and Odd(x) is the odd integer nearest to x. For $w = 16$, P_{16} equals $(\text{b7e1})_{16}$ or $(1011011111100001)_2$, and Q_{16} equals $(\text{9e37})_{16}$ or $(1001111000110111)_2$.

As discussed in [1], the key-expansion algorithm consists of three simple algorithmic parts, see Listing 3. The first step is to copy the secret key $K[0: b-1]$ into an array $L[0: c-1]$, where b is the number of bytes in the secrete key, c is the number of words in the secrete key ($c = \lceil b/u \rceil$), and u is the number of bytes per word. Note that any unfilled byte

positions of L are zeroed. The second step is to initialize array S to a particular fixed (key-independent) pseudo-random bit pattern, using an arithmetic progression modulo 2^w determined by the "magic constants" P_w and Q_w, where $S[0] = P_w$ and $S[i] = S[i-1] + Q_w$, for $i = 1$ to $t - 1$. Finally, the third step of key-expansion is to mix in the user's secret key in three passes over the arrays S and L. More precisely, due to the potentially different sizes of S and L, the larger array will be processed three times, and the other may be handled more times.

Listing 3: Key-expansion algorithm

```
// First step
    for i = b – 1 downto 0
        L[i / u] = (L[i / u] <<< 8) + K[i];
// Second step
    S[0] = P_w
    for i = l to t – l
        S[i] = S[i – 1] + Q_w
// Third step
    i = j = 0
    A = B = 0
    do 3*max(t, c) times
        A = S[i] = (S[i] + A + B) <<< 3
        B = L[j] = (L[j] + A + B) <<< (A + B)
        i = (i + 1) mod(t)
        j = (j + 1) mod(c)
```

In contrast to encryption and decryption modules, the key-expansion module is implemented as rolled since it is expanded only once regardless on the size of data. In this paper, the key-expansion unit accepts 96-bit user's secrete key to generate round keys needed for encrypting/decrypting data. According to the key-expansion module shown in Listing 3, the number of iterations for the key-expansion module equals 56 since t (=18) is greater than c (=6) and the number of iterations for the key-expansion module equals $3*t$ (=56). Thus, uploading new secrete key requires 56 clock cycles for filling the expanded key array S. Table 3 presents the statistics of hardware implementation of the key-expansion module on Virtex-5 XC5VLX110T. The number of LUT flip-flop pairs used is 1172, where the numbers with unused flip-flops, unused LUTs, and fully used LUT-FF pairs are 450, 577, and 145, respectively.

Table 3: HDL synthesis of the key-expansion unit

Macro Statistics	16-bit adder: 4 3-bit up counter: 1 5-bit up counter: 1 7-bit up counter: 1 1-bit register : 1 16-bit register: 44
Slice Logic Utilization	Number of Slice Registers: 722 Number of Slice LUTs: 595 Number used as Logic: 595
Slice Logic Distribution	Number of LUT Flip Flop pairs used: 1172 Number with an unused Flip Flop: 450 Number with an unused LUT: 577 Number of fully used LUT-FF pairs: 145
IO Utilization	Number of IOs: 388 Number of bonded IOBs: 388
Timing (ns) Summary	Minimum period: 8.422 (Maximum Frequency: 118.733MHz) Minimum input arrival time before clock: 2.724 Maximum output required time after clock: 2.775

3. THE ARCHITECTURE OF CryptoVVSHP

VVSHP has a modified five-stage pipeline for executing multi-scalar/vector instructions by: (1) fetching 128-bit VLIW instruction, (2) decoding/reading operands of four individual instructions, (3) executing four scalar/vector operations, (4) accessing memory to load/store 128-bit data, and (5) writing back up to four results [7]. Figure 4 shows that CryptoVVSHP extends VVSHP by encryption/decryption/key-expansion units based on RC5 cryptographic algorithm to process encrypted scalar/vector data. CryptoVVSHP uses 128-bit (4×32-bit) VLIW for encoding scalar/vector instructions. All instructions are fixed length VLIW, which simplifies the instruction decoding hardware. CryptoVVSHP instruction formats (R-format, I-format, and J-format), which are very close to MIPS [9]. The first instruction in VLIW can be either scalar or vector instruction. However, the remaining slots in VLIW must be scalar instructions. This simplifies the

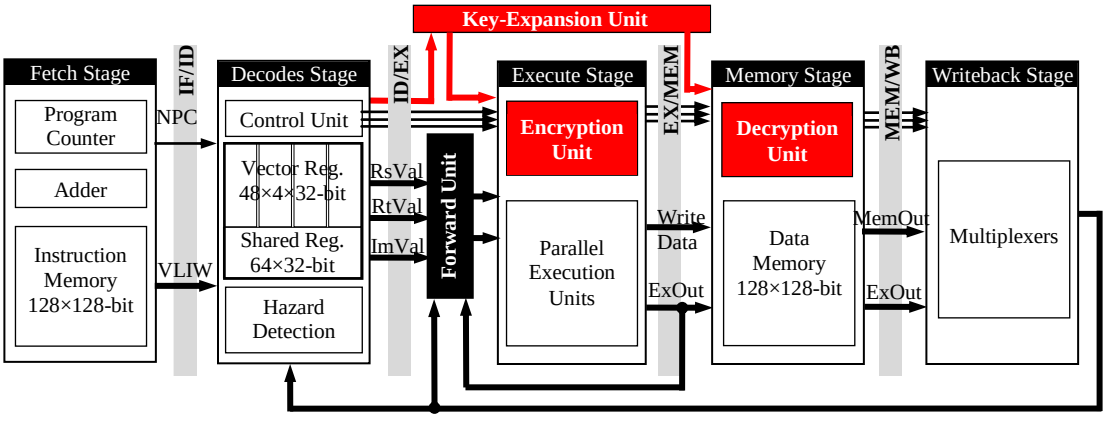


Fig. 4. Block diagram of CryptoVVSHP.

implementation of CryptoVVSHP and does not effect on the performance since a vector instruction can encode multiple operations. In addition to load (LW) and store (SW) instructions, CryptoVVSHP adds new instructions called load-with-decryption (LWD), store-with-encryption (SWE), and key-expansion (KeyEx) for processing encrypted scalar/vector data.

In more details, Figure 5 shows the CryptoVVSHP datapath for processing encrypted scalar/vector data. The VLIW instruction pointed by the program counter (PC) is read from the instruction memory of the fetch stage and stored in the instruction fetch/decode (IF/ID) pipeline register. The control unit in the decode stage reads the fetched VLIW instruction from IF/ID pipeline register and vector length register (VLR) to generate the proper control signals needed for processing multi-scalar/vector data. Three new control signals called EncVld, DecVld, and KeyVld are generated by decoding SWE, LWD, and KeyEx instructions to activate the encryption, decryption, and key-expansion units, respectively. Like VVSHP, the register file of the CryptoVVSHP has two parts: (1) shared scalar-vector part with eight-read/four-write ports 64x32-bit registers (64 scalar or 16×4 vector registers) for storing scalar/vector data, and (2) vector part with two-read/one-write ports 48 vector-registers, each stores 4×32-bit vector data. It can process vector data stored in multiple registers with lengths vary from 1 to 256. The register file can be seen as 64×32-bit scalar registers or 64×4×32-bit vector registers.

The execution units of CryptoVVSHP operate on the 128-bit operands prepared in the decode stage (RsVal, RtVal, and ImVal) and perform operations specified by the control unit, which depends on opcode/function fields of each individual instruction in VLIW. For load/store instructions, the first execution unit adds 32-bit $RsVal_1$ and 32-bit $ImVal_1$ to form the effective address. In addition, the $RtVal_1$, $RtVal_2$, $RtVal_3$, and $RtVal_4$ are encrypted when EncVld control signal is asserted by SWE instruction. For register-register instructions, the execution units perform the operations specified by the control unit on the operands fed from the register file ($RsVal_1$/$RtVal_1$, $RsVal_2$/$RtVal_2$,

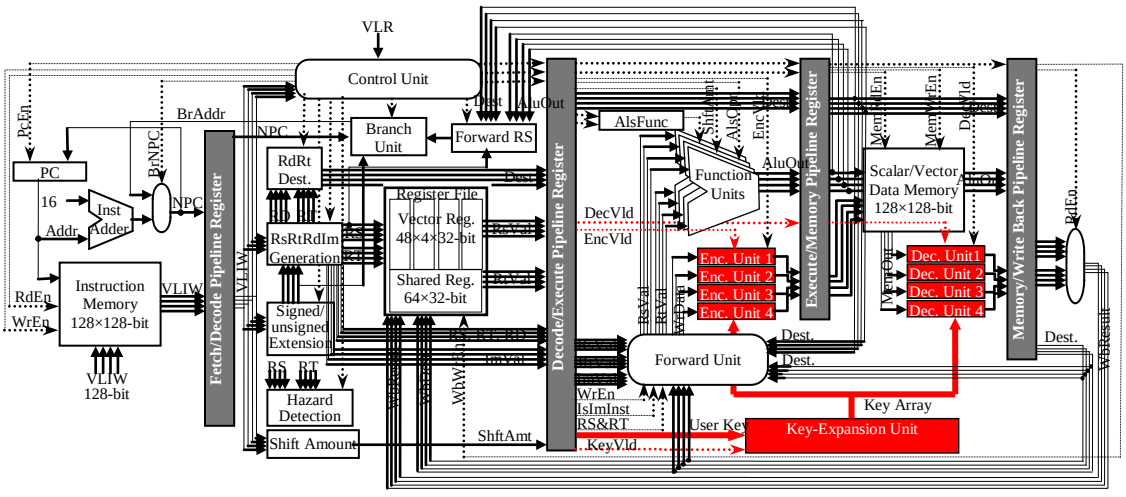


Fig. 5. CryptoVVSHP datapath for executing encrypted scalar/vector data.

$RsVal_3$/$RtVal_3$, and $RsVal_4$/$RtVal_4$) through ID/EX pipeline register. For register-immediate instructions, the execution units perform the operations on the source values ($RsVal_1$, $RsVal_2$, $RsVal_3$, and $RsVal_4$) and the extended immediate values ($ImVal_1$, $ImVal_2$, $ImVal_3$, and $ImVal_4$). In all cases, the results of the execution units are placed in the EX/MEM pipeline register.

The CryptoVVSHP registers can be loaded/stored individually using load/store instructions. Displacement addressing mode is used for calculating the effective address in the execute stage. Four contiguous elements (128-bit) can be loaded/stored per clock cycle from/to data memory. The output of data memory (4×32-bit MemOut) are decrypted when DecVld control signal is asserted by LWD instruction.

Finally, the writeback stage of CryptoVVSHP stores the 4×32-bit results that come from the decryption units or from the execution units into the CryptoVVSHP register file. Depending on the effective opcode of each individual instruction in VLIW, the register destination field is specified by either RD or RT using RdRtDest unit. The control signals 4×1-bit WbWrEn are used for enabling the writing 4×32-bit results into the CryptoVVSHP register file

Note that CryptoVVSHP has common datapaths for executing multi-scalar/vector instructions. This increases the efficiency of hardware and makes efficient exploitation of resources even though the percentage of data parallelism is low. Instruction memory of size 128×128-bit stores 128-bit VLIW instructions of an application, where each VLIW has four-scalar instructions or a vector instruction concatenating with three no-operations. Data memory of size 128×128-bit loads/stores scalar/vector data needed for processing multi-scalar/vector instructions. The first part of the register file (64×32-bit registers) is used for both multi-scalar/vector elements. The control unit feeds the unified execution units by the required operands (scalar/vector elements) and can produce up to four results each clock cycle. The writeback stage writes into the register file up to 4×32-bit scalar/vector results per clock cycle coming from the decryption units or from the execution units. Besides, the encryption, decryption, and key-expansion units are used for processing encrypted scalar/vector data.

4. FPGA IMPLEMENTATION OF CryptoVVSHP

The design of our proposed CryptoVVSHP processor is implemented using VHDL targeting the Xilinx FPGA Virtex5, XC5VLX110T-3FF1136 device [8]. Virtex-5 XC5VLX110T has 160×54 array of configurable logic blocks (8640 CLBs). A single Virtex-5 CLB comprises two slices, with each containing four 6-input look-up tables (LUTs) and four flip-flops (FFs), for a total of eight 6-input LUTs and eight FFs per CLB. Thus, Virtex-5 XC5VLX110T has 17280 slices (69120 6-input LUTs and 69120 FFs). For memory resources, it has 1120 Kbits distributed RAM and 148 block RAMs.

Each block RAM is 36 Kbits in size and can be used as two independent 18-Kbit blocks. Moreover, Virtex-5 XC5VLX110T has 64 DSP48E slices, each contains a 25×18 multiplier, an adder, and an accumulator. Note that Virtex-5 family is the first FPGA platform to offer a real 6-input LUT with fully independent (not shared) inputs. By properly loading LUT, any 6-input arithmetic/logical/ROM function can be implemented. In addition to this, some slices called SLICEM support two additional functions: storing data using distributed RAM and shifting data with 32-bit registers. See [8] for more details.

Figure 6 shows the statistics of the FPGA implementation of CryptoVVSHP, which includes the following components.

- Fetch stage (program counter, instruction address adder, instruction memory, and fetch outputs) requires 16576 slice registers and 9696 slice LUTs, where the total number of LUT-FF pairs used is 26247: 9671 with unused FFs, 16551 with unused LUTs, and 25 of fully used LUT-FF pairs.
- Decode stage (control unit, register file, hazard detection unit, forward RS, shift amount, sign extension, RsRtRdIm generation, branch unit, and decode outputs) requires 8910 slice registers and 20510 slice LUTs, where the total number of LUT-FF pairs used is 28848: 19938 with unused FFs, 8338 with unused LUTs, and 572 of fully used LUT-FF pairs.
- Execute stage (four arithmetic/logical/shift units, four encryption units, arithmetic/logical/shift functions, and execute outputs) requires 1830 slice registers and 15592 slice LUTs, where the total number of LUT-FF pairs used is 16563: 14733 with unused FFs, 971 with unused LUTs, and 859 of fully used LUT-FF pairs.
- Memory stage (data memory, four decryption units, and memory outputs) requires 16800 slice registers and 21578 slice LUTs, where the total number of LUT-FF pairs used is 38122: 21322 with unused FFs, 16544 with unused LUTs, and 256 of fully used LUT-FF pairs.

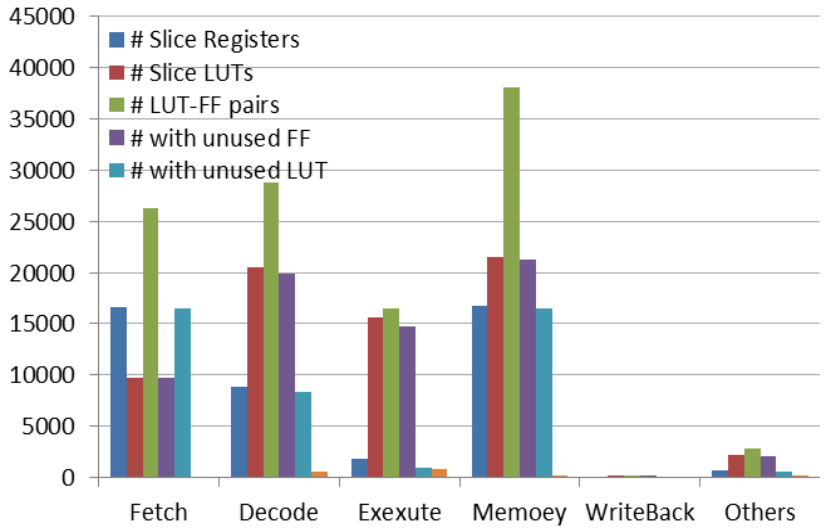


Fig. 6. Statistics of the FPGA implementation of CryptoVVSHP.

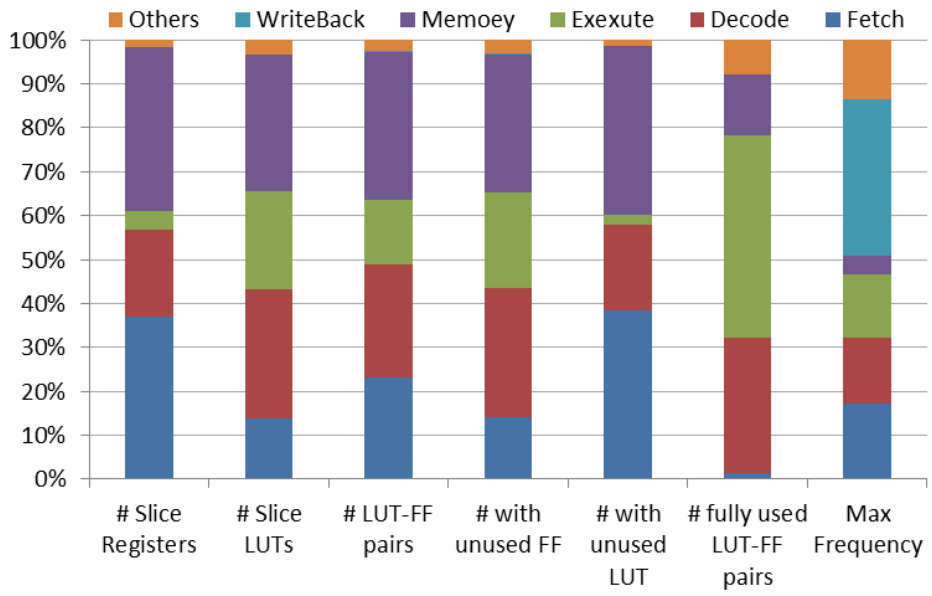
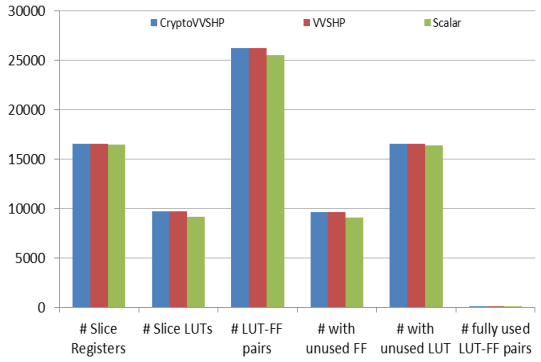


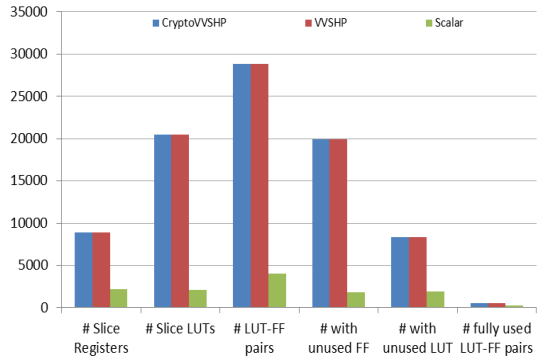
Fig. 7. Percentage of complexity in each stage of CryptoVVSHP.

- Writeback stage (multiplexers) requires 159 slice LUTs, where the total number of LUT-FF pairs used is 159 (159 with unused FFs).
- Others (key-expansion unit, forward unit, and RdRtDest unit) requires 722 slice registers and 2240 slice LUTs, where the total number of LUT-FF pairs used is 2817: 2095 with unused FFs, 577 with unused LUTs, and 145 of fully used LUT-FF pairs.

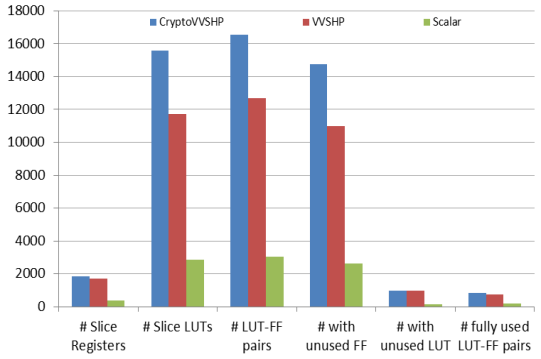
Figure 7 shows the percentage of fetch, decode, execute, memory, writeback, and others in the overall implementation of CryptoVVSHP. Note that the complexity in LUT-FF pairs of the memory stage is the highest because it has 128×128-bit data memory and four decryption units. The order of CryptoVVSHP stages with respect to complexity are memory access (33.8%), decode (25.6%), fetch (23.3%), execute (14.7%), writeback (0.1%), and others (2.5%). Moreover, Figure 8 compares the implementation of CryptoVVSHP with the baseline scalar (five-stage pipeline [10]) and VVSHP processors. The complexity of the CryptoVVSHP is about 207% and 123% of the baseline scalar and VVSHP processors, respectively.



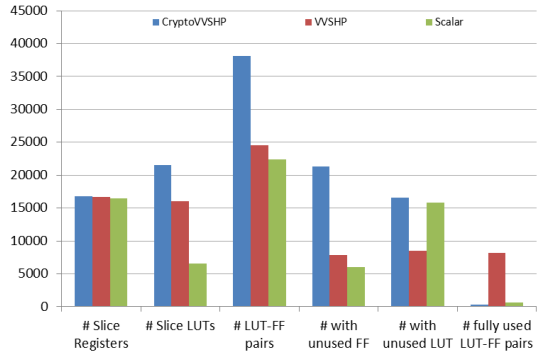
(*a*) Fetch stage



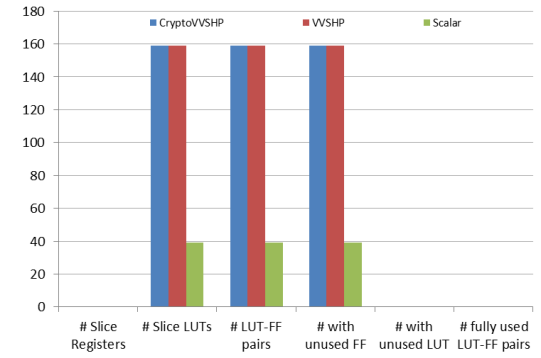
(*b*) Decode stage



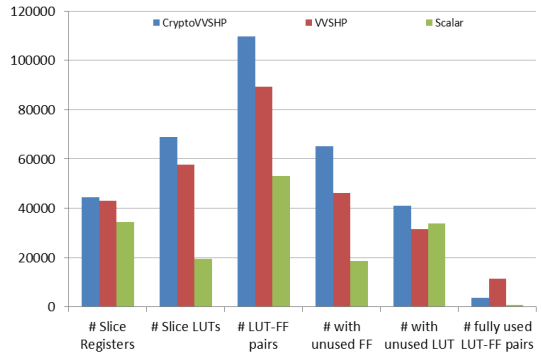
(*c*) Execute stage



(*d*) Memory access stage



(*e*) Writeback stage



(*f*) Overall pipeline

Fig. 8. Synthesizing CryptoVVSHP, VVSHP, and scalar pipelines on FPGA Virtex-5.

5. CONCLUSION

This paper modifies the architecture of VVSHP to process encrypted scalar/vector data. CryptoVVSHP allows scalar/vector data to exist in decrypted form internally, however, decrypted data is not externally accessible. The load/store instructions automatically cause the appropriate decryption/encryption operations to be performed on scalar/vector data. Thus, after loading encrypted data from memory to register file, they

are decrypted. Moreover, before storing data from register file to memory, they are encrypted. In few words, data is decrypted only while in the pipeline for processing.

This paper shows the FPGA implementation of our proposed CryptoVVSHP on Virtex-5 XC5VLX110T, which requires 44559 slice registers and 68833 slice LUTs, where the total number of LUT-FF pairs used is 109737: 65178 with unused FFs, 40904 with unused LUTs, and 3655 of fully used LUT-FF pairs. The complexity of the CryptoVVSHP is about 207% and 123% of the baseline scalar and VVSHP processors.

In the future, other cryptography algorithms like the advanced encryption standard (AES) can be considered as a security extension to scalar/vector processors. Like multimedia extensions, the instruction set architecture of scalar/vector processors can be extended with security instructions that are executed on extended security hardware.

REFERENCES

[1] R. Rivest, "The RC5 Encryption Algorithm," MIT Laboratory for Computer Science, http://people.csail.mit.edu/rivest/pubs/Riv94.pdf, 1997.

[2] R. Rivet, L. Adleman, and M. Dertouzos, "On Data Banks and Privacy Homomorphisms," Foundation of Secure Computation, Academic Press. New York, pp. 169-179, 1978.

[3] C. Sagedy, "ECEC 490: Processor Design Project Page," http://chris.sagedy.com/projects/ecec490_fa08/#encrypted, December 2008.

[4] N. Ahituv, Y. Lapid, and S. Neumann, "Processing Encrypted Data," Communications of the ACM, Vol. 30, No. 9, pp. 777-780, September 1987.

[5] C. Gentry, "A Fully Homomorphic Encryption Scheme," Ph.D. Thesis, Department of Computer Science, Stanford University, September 2009

[6] B. Weir, "Homomorphic Encryption," Master Thesis, Combinatorics and Optimization Department, University of Waterloo, Canada, 2013.

[7] M. Soliman "Merging VLIW and Vector Processing Techniques for a Simple, High-Performance Processor Architecture," Submitted to Microelectronics Journal in May 2014, Revised in October 2014.

[8] Virtex-5 FPGA User Guide, UG190 (v5.4). http://www.xilinx.com/support/documentation/user_guides/ug190.pdf, March 2012,

[9] G. Kane, MIPS RISC Architecture (R2000/R3000), Prentice Hall, 1989.

[10] D. Patterson and J. Hennessy, Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufman, San Francisco, CA, 5th Edition, October 2013.