# On the Design and Implementation of a RISC Processor Extension for the KASUMI Encryption Algorithm

Tomas Balderas-Contreras, Rene Cumplido, Claudia Feregrino-Uribe
Computer Science Department
National Institute for Astrophysics, Optics and Electronics, INAOE
Luis E. Erro 1, Tonantzintla, Puebla 72840, Mexico
rcumplido@inaoep.mx

## Abstract

Modern cellular networks allow users to transmit information at high data rates, have access to IP-based networks deployed around the world, and access to sophisticated services. In this context, not only is it necessary to develop new radio interface technologies and improve existing core networks to reach success, but guaranteeing confidentiality and integrity during transmission is a must. The KASUMI block cipher lies at the core of both the *f8* data confidentiality algorithm and the *f9* data integrity algorithm for Universal Mobile Telecommunications System networks. KASUMI implementations must reach high performance and have low power consumption in order to be adequate for network components. This paper describes a specialized processor core designed to efficiently perform the KASUMI algorithm. Experimental results show two orders of magnitude performance improvement over software only based implementations. We describe the used design technique that can also be applied to implement other Feistel-like ciphering algorithms. The proposed architecture was implemented on a FPGA, results are presented and discussed.

***Keywords***

Cryptography; Block cipher; KASUMI; Hardware implementation; RISC Processors

## 1. Introduction

The nature of the information that flows throughout modern cellular communications networks has evolved noticeably since the early years of the first generation systems, when only voice sessions were possible. With today's networks it is possible to transmit both voice and data, including e-mail, pictures and video. The importance of the security issues is higher in current cellular networks than in previous systems because users are provided with the mechanisms to accomplish very crucial operations like banking transactions and sharing of confidential business information, which require high levels of protection. Weaknesses in security architectures allow successful eavesdropping, message tampering and masquerading attacks to occur, with disastrous consequences for end users, companies and other organizations.

Not only does the Universal Mobile Telecommunications System (UMTS) standard provide advanced communications services, it also includes the means to guarantee high levels of confidentiality and integrity of information as well as the authentication of each entity engaged in a communications session. The answer to the security challenge is the development of a sophisticated mutual authentication protocol [1], and the *f8* confidentiality algorithm and the *f9* integrity algorithms based on the KASUMI block cipher [2, 3, 4].

As stated in [5], solutions to implement lightweight cryptology are strongly needed. This becomes more relevant in mobile applications, where performance and cost are particularly important issues. There are basically three types of software and hardware platforms to implement cryptographic algorithms: a) Software implementations using general purpose processors, b) embedded software implementations (such as smart cards), and c) implementations using dedicated hardware (ASIC, FPGA). The authors in [5] also state that one possible solution that combines the flexibility of software-based implementations and the performance of custom hardware architectures to the efficient implementation of cryptographic algorithms is to develop optimized processor architectures by customizing both the processor's instruction set and its micro-architecture. Thus, on the basis that some algorithms implemented in hardware achieve higher performance than the corresponding software codification and that software allows implementing complex functions, this paper describes a solution strategy to the problem of the efficient implementation of the KASUMI-based UMTS security functions by means of a combination of hardware and software modules. At a glance, the experimentation work is based on the hypothesis that it is possible to implement in hardware the most performance demanding component of both the *f8* and *f9* algorithms, i.e. the KASUMI block cipher; attach this hardware module as a functional unit to a general purpose processor, extending its instruction set to exploit the new hardware, and then build the whole algorithms in software.  Results show two orders of magnitude performance improvement over software only based implementations. The results also show that the proposed extension achieves the higher throughput/area ratio that other reported architectures due to the efficient reutilization of the optimized two-round ciphering datapath. Thus, the proposed architecture offers a good alternative for implementing the KASUMI algorithm.

## 2. UMTS' security architecture

According to the specifications, the security architecture is made up of a set of security features and security mechanisms [1]. A security feature is a service capability that meets one or several security requirements. A security mechanism is an element or process that is used to carry out a security feature. Two of the most important UMTS´s security features are: Integrity and confidentiality. The algorithm defined to perform the confidentiality tasks is called *f8*. The mechanism that carries out the integrity security feature is based on an UMTS Integrity Algorithm (UIA) which in turn is based on the *f9* algorithm.

Figure 1 illustrates the structure of the *f8* algorithm. Several instances of the KASUMI block cipher are organized in a so called Output Feedback (OFB) mode [6]. Each block cipher generates 64 bits of the whole output keystream and forwards its output to the input of the following block cipher, subject to its modification by a XOR operation with a counter and a static value.
The input parameter LENGTH indicates the length of both the keystream and the plaintext stream. The parameter BEARER identifies to each radio bearer among those associated with each user; this input value avoids the use of the same keystream for encryption/decryption in every radio bearer.

Figure 2 shows the internal structure of the *f9* algorithm. It is based on a series of KASUMI block ciphering modules interconnected in a variant of the Cipher Block Chaining (CBC) mode [6]. The algorithm combines the 64-bit intermediate outputs of all of the block ciphers by using XOR operations, and, lastly, applies the KASUMI algorithm to this sum. The 64-bit output of this final process is truncated to 32 bits to obtain the MAC-I value.

## 3. KASUMI algorithm description

KASUMI is based on a previous block cipher called MISTY1 [3]. MISTY1 was chosen as the foundation for the 3GPP ciphering algorithm due to its proven security against the most advanced methods to break block ciphers, namely cryptanalysis techniques. In addition, MISTY1 was heavily optimized for hardware implementation. KASUMI has a Feistel structure and operates on 64-bit data blocks under control of a 128-bit encryption key K [1]. KASUMI has the following features: it performs 8 rounds of ciphering, the input plaintext is the input to the first round, ciphertext is the last round's output, K is used to generate a set of round keys {KLi, KOi, KIi} for each round I, each round computes a different function, as long as the round keys are different, and the same algorithm is used both for encryption and decryption.

As can be seen in Figure 3, the KASUMI block cipher has a different setup for the functions within rounds. For odd rounds the round-function is computed by applying the FL function followed by the FO function. For even rounds FO is applied before FL. FL, shown in Figure 3.d, is a 32-bit function made up of simple

AND, OR, XOR and left rotation operations. FO, depicted in Figure 3.b, is also a 32-bit function having a three-round Feistel organization which contains one FI block per round. FI, see Figure 3.c, is a non-linear 16-bit function having itself a four-round Feistel structure; it is made up of two 9-bit substitution boxes (S-boxes) and two 7-bit S-boxes. Figure 3.c shows that data in the FI function flow along two different paths: a nine-bit long path (thick lines) and a seven-bit path (thin lines). Notice that in Feistel structures, such as the ones used in this algorithm; each round's output is twisted before being applied as input to the following round. After completing eight rounds KASUMI produces a 64-bit long ciphertext block corresponding to the input plaintext block.

Several proposals have been published previously that use different approaches to implement KASUMI in hardware, ranging from reuse techniques, addition of internal registers to reduce critical path and pipelined designs [7-13]. After analyzing these proposals it is clear that there is an important and unavoidable tradeoff between performance and complexity in terms of area, thus the goals of this work are twofold. First, to reach a good balance between high performance and low complexity during the implementation of KASUMI and second, to present the methodology used to design a specialized functional unit attached to a RISC processor.

## 4. Proposed architecture and hardware implementation

An extension was made to a MIPS-based processor core to support block ciphering according to the KASUMI algorithm. A new functional unit was added to the datapath and four specialized new instructions were defined to control the extended unit. To prove this concept, the use of a complex core to carry out the job is feasible, but the time needed to understand the core's internals and conceive a way to extend it increases noticeably, thus the simplicity of the source code was a key factor to make the final decision. The processor core chosen for this work is the MyRISC core [14], which models a MIPS R2000 32-bit processor with a five-stage pipeline structure. It is important to mention that this same approach can be used to integrate the proposed extension to other RISC type processors.

## *4.1 Processor organization*

Figure 4 shows the organization of the proposed extension to the MyRISC core. The part above the thick horizontal line corresponds to the initial RISC processor core as it is distributed. The components lying below the thick line correspond to the KASUMI extension, which carries out the processing corresponding to two rounds. The modules that store and generate data operands are located in the processor's Instruction Decode (ID) stage, whereas the modules that perform encryption operations belong to the Execute (EX) stage.

## *4.2 The extended register file*

The new functional unit contains ten 32-bit registers that store the data it processes. Extended instructions move data from/to integer registers to/from a register within this new register file. Figure 5 shows the organization of this data unit. Registers 0 and 1 store the plaintext block the KASUMI functional unit works with; after the ciphering process the registers store the ciphertext block produced.

The 32 most significant bits of the block are stored in register 0, whereas register 1 stores the 32 least significant bits. The 128-bit encryption key K is split into four 32-bit parts and stored in registers 2 to 5. Registers 6 to 9 store the ciphering constants used along with the encryption key to generate the set of round keys KLi,KOi,KIi for each round i. There is no need to preload the array of constants since these values are automatically stored every time the RESET signal is asserted.

Any of the first six registers within the extended register file can be synchronously written by specifying its address and the value to store, in the same way as for integer registers. Registers 0 and 1 can be written in parallel to store the ciphertext block produced by the block ciphering modules. These two kinds of writing can not be accomplished simultaneously. The array that stores the encryption key K (registers 2 to 5) is synchronously rotated upwards to compute the appropriate round keys for the next two rounds. This is also true for the array that stores the ciphering constants (registers 6 to 9). The only kind of writing allowed occurring at the same time as the rotation of the arrays is the parallel writing of registers 0 and 1. The register file asynchronously outputs the contents of the ten internal registers. An additional output issues the contents of a specific register indicated by an input address line in an asynchronous fashion as well.

## *4.3 The forwarding unit*

This module allows the KASUMI functional unit to use correct and up-to-date values of the plaintext block and the encryption key. The extended processor allows different instructions that modify the first six registers to be executing along the pipeline. The forwarding unit receives values from the KASUMI register file and from different pipeline stages and determines if the values stored in registers are old, in which case the unit outputs the new values before they are actually written in the extended register file. This unit makes its decision based on input control signals and register address lines coming from either the stages in the integer pipeline or the modules comprising the KASUMI functional unit. The forwarding unit outputs the plaintext block sent to the extended register file most recently, or the ciphertext block computed most recently, directly to the ID/EX pipeline register. The outputs corresponding to the encryption key are used to generate the next two sets of round keys.

## *4.4 The ciphering datapath*

This module is parallel to the EX stage of the processor's datapath and performs the encryption process using the block issued by the forwarding unit and the round keys computed by the key generation unit. It carries out an odd round followed by an even round of the KASUMI algorithm in four steps: K1, K2, K3 and K4, where each step takes one clock cycle to complete.

The strategy followed to design this module is shown in figure 6. The manipulation strategy considers a pair of consecutive rounds; an odd round followed by an even round. It changes the structure of the pair without altering its effects, adds components that balance the structure and discovers a design pattern that replicates. This pattern then turns into the basic building block that is implemented once and then reused until completion of the ciphering process. Figure 6.a shows two reordered FO blocks, while figure 6.b shows the result of expanding the two FO blocks and splitting the 32-bit XOR gate located between the two FO blocks into two 16-bit XOR gates and “unfolding” the datapath comprising the upper FO function block’s output, the two 16-bit XOR gates and the lower FO function block. Notice that both, the 32-bit R0 input and the 32-bit R2 output, are now split into two 16-bit lines. Figure 6.c shows the result of joining the two FO function blocks to highlight the parallelism between each pair of FI function blocks. Some 16-bit XOR gates with one zero input are added along the datapath in certain places so that the datapath can be divided in three structurally similar sections each with two PI blocks. The final ciphering datapath for the two rounds is shown in figure 7, where each pair of PI blocks from figure 6.c is grouped into the so called dpFI blocks. This reordering process was originally proposed in [7], were a special purpose reuse-based architecture for implementing the KASUMI algorithm was described. The architecture exploits reutilization of components to implement the eight rounds required by the algorithm.

In spite of this multicycle operation, the ciphering datapath is not intended to work in a pipelined fashion. This means that an instruction that uses the ciphering datapath is not allowed to enter the K1 module until the previous instruction has left the K4 module. In the KASUMI functional synchronous registers are indicated by grey boxes. When the ciphering process reaches the K3 module it commands the KASUMI register file to rotate the arrays storing the encryption key and the ciphering constants, by means of a control signal indicated by a dashed line in figure 4, in order for the next two sets of round keys to be available in the next two clock cycles, when a new instruction enters K1. During the K4 step the corresponding module bypasses the computed ciphertext block to the forwarding unit to override the block stored in registers and make the new one available as the plaintext block to process in the next clock cycle. A control signal indicated by a dashed line is also bypassed to help the forwarding unit to determine the correct value of the block. When the ciphering instruction leaves the K4 module it enters the pipeline’s memory access stage (MEM) where, in the case of KASUMI instructions, the ciphertext block just computed is actually written into registers 0 and 1 within the extended register file. Meanwhile, a new KASUMI ciphering instruction can start with the K1 step.

## *4.5 The key generation unit*

The key generation unit outputs two sets of round keys ({KL1,KO1,KI1} and {KL2,KO2,KI2}) and stores them into the ID/EX pipeline register to be issued to the ciphering datapath during the next clock cycle. This unit receives as inputs the four 32-bit words comprising the encryption key K from the forwarding logic and the four 32-bit words storing the ciphering constants from the extended register file. Figure 8 shows the organization of the key generation unit.

## *4.6 The extended instructions*

Four instructions were added to the MIPS instruction set to control the extended KASUMI functional unit. Instruction formats for the four instructions are shown in Figure 9. The instructions are described next.

## The *kxor1* instruction

It carries out the operation Rs⊕Rt, where Rs and Rt are integer registers. This instruction uses the integer EX and MEM pipeline stages and saves the result in the extended KASUMI register file at the entry addressed by the four least significant bits of KRd during the integer WB stage. Its mnemonic is
*kxor1* KRd, Rs, Rt.

## The *kxor2* instruction

It carries out the operation Rs ⊕ KRt, where Rs is an integer register and KRt is a register in the extended KASUMI register file. This instruction uses the integer EX and MEM pipeline stages and saves the result in the extended KASUMI register file at the entry addressed by the four least significant bits of KRd during the integer WB stage. Its mnemonic is *kxor2* KRd, Rs, KRt.

## The *kxor3* instruction

It carries out the operation Rs ⊕ KRt, where Rs is an integer register and KRt is a register in the extended KASUMI register file. This instruction uses the integer EX and MEM pipeline stages and saves the result in the integer register addressed by Rd during the integer WB stage. Its mnemonic is *kxor3* Rd, Rs, KRt.

## The *k2rnd* instruction

It carries out the operations corresponding to a sequence of an odd round and an even round of the KASUMI block cipher. It does not need explicit operands; it uses the outputs of the forwarding logic and the key generation unit. A sequence of four *k2rnd* instructions performs the whole KASUMI algorithm. *k2rnd* is a multicycle instruction whose execution phase is actually made up of four cycles: K1, K2, K3 and K4. Only after a *k2rnd* instruction has finished with cycle K4, the next *k2rnd* instruction will enter K1. During the MEM stage this instruction issues the computed block to the extended register file in order for it to be stored in registers 0 and 1. Since this operation is synchronous, the block is actually written when the instruction enters the WB stage. Its mnemonic is *k2rnd*.

## *4.7 Details about the execution of extended instructions*

Figure 10 illustrates the pipelined execution of the instructions making up the encryption process. The operands of the instructions are carefully chosen to show how the extended processor deals with special execution conditions. The first six *kxor1* instructions load the plaintext block and the encryption key into the extended registers. The next four *k2rnd* instructions perform the encryption process using the operands stored by the previous instructions. Notice that the address of the target register in instruction 1, which is 0, equals the address of the first source register in instruction 2. For integer instructions this would cause a data hazard and the bypassing of the value computed by instruction 1 in the EX stage to the ID stage of instruction 2 during the third clock cycle. However, for the instructions in figure 9 the bypassed value is ignored by the integer forwarding logic since the target register of instruction 1 is an extended register, not an integer register as the source register of instruction 2. This situation is called a false data hazard and is handled by the processor by appropriately setting a control signal. The same situation occurs during cycles 4 and 5. A true data hazard occurs during the eighth cycle because the first *k2rnd* instruction needs to compute the two sets of round keys and, at this time, the encryption key K has not been completely stored. However, the forwarding logic in the KASUMI functional unit overcomes this problem. This module receives the bypassed data signals from the *kxor1* instructions in the EX, MEM and WB stages of the integer pipeline and issues them to the key generation unit to produce the round keys needed in the next cycle. The KASUMI forwarding logic ignores any bypassed signal issued by an instruction different from *kxor1* and *kxor2*. Note that the overlapped execution of a *k2rnd* instruction (10) and an integer instruction (11) is allowed. This situation does not produce structural hazards since the integer MEM and WB stages do not share any module with the corresponding MEM and WB stages in the extended functional unit. When a *k2rnd* instruction, e.g. the instruction 7, enters K1 a no-operation integer instruction enters EX in the integer portion of the processor's pipeline.  The number of cycles elapsed since instruction 7 starts execution until instruction 10 leaves the execution stage is 16 cycles.

## 5. Performance evaluation

The number of instructions needed to implement the KASUMI block cipher in software, using the standard MIPS32 instruction set, is much higher than the number of extended instructions needed when the proposed extension is used. The C code for KASUMI included in [3] is made up of five functions: FI(), FO(), FL(), KeySchedule() and Kasumi(). This code is suitable to carry out a thorough study concerning the number of instructions executed by a compiled program. The source code is compiled using the C cross-compiler provided by the Software

Development Environment (SDE) for MIPS-based products toolkit from MIPS Technologies [15], which is actually a built of GNU's C compiler. The compiler is instructed, with the -Os option, to enable all the optimizations intended to reduce code size and generate the shortest executable program. This executable program is then disassembled using the objdump utility, which displays information from object files and is part of the GNU's set of binary utilities (Binutils).

The result of the study is summarized in tables 1-5. The instructions making up each module are counted, special constructs like loops, control transfer statements and function calls are identified and a precise counting of executed instructions is carried out for each of these constructs. The last entry of the tables provides the exact number of instructions the MIPS processor executes for the corresponding function. The number of instructions required to perform the KASUMI algorithm is given by adding the counts for the top level modules Kasumi() and KeySchedule(), i.e. 1540 + 915 = 2455 instructions. For the proposed approach, the number of cycles elapsed since instruction 7 starts execution until instruction 10 leaves the execution stage is 16 clock cycles. A total number of 26 clock cycles are needed to carry out the whole ciphering process including the storage of the plaintext block and the encryption key into the extended register file. In spite of the transfer overhead, that can be avoided by exploiting instruction reordering; this approach allows reducing the number of instructions required by two orders of magnitude when compared with a software only implementation, at the expense of the addition of a compact functional unit. The proposed extension was implemented using a Virtex-II FPGA device from Xilinx. It requires 448 slices and can operate at a maximum frequency of 96.33 MHz. Thus, considering a latency of 16 clock cycles and assuming that the processor can operate at the same frequency that the extension unit, the proposed architecture can achieve a throughput of 385 Mbs.

Other architectures for the KASUMI algorithm have been reported. In [9], the authors report two architectures that implement logic for only one round, i.e. the FO and the FL function blocks. The first architecture, called Type 1, iterates over these two components eight times until completion of the process, feeding the design's output back to its input at the end of each iteration, sacrificing performance in the interests of achieving low hardware complexity. The Type 2 architecture contains a four-stage inner-round pipelined FO module that results in an increased operating frequency and an improved throughput, by a factor of four. The two-round architecture described in [10] takes advantage of both inner- and outer-round pipeline techniques to decrease the period of the clock and increase the throughput. Inner-round registers are negative edge-triggered, whereas outer-round registers are positive edge-triggered; consequently, the execution time of each round is one clock cycle. The pipelined design allows this circuit to process two blocks simultaneously, with an initial latency of eight cycles. The S-boxes in this architecture are implemented with combinational logic. The two architectures reported in [11] are similar to that described in [10]. The authors look to reduce the area required by implementing a two-round iterative architecture. An interesting fact about this design is that the S7 and S9 S-boxes are implemented as combinational logic and, alternatively, mapped to embedded memory blocks within the FPGA. There are registers at the end of each round, making the architecture to have a total completion latency of 8 clock cycles when the S-boxes are implemented as combinational modules, and 40 cycles when the S-boxes are mapped to embedded memory blocks; this due to the inner-round pipeline stages introduced by the registered outputs of the synchronous memory blocks. This two-round design is not intended to work in a pipelined fashion. It is possible to manipulate the structure of the KASUMI block cipher, by means of aggressive simplifications, to get inexpensive datapaths with long latencies that carry out the ciphering process. The work reported in [12] presents the application of a simplification technique to design two KASUMI architectures with latencies of 56 and 32 cycles, respectively. A third architecture with a latency of 8 cycles is mentioned, and its results provided, but the architecture is not fully described. A crypto processor that consists of a 32-bit RISC processor block and coprocessor blocks dedicated to the AES, KASUMI, SEED, triple-DES private key crypto algorithms and ECC and RSA public key crypto algorithm is described in [13]. The 32-bit RISC type processor controls the dedicated crypto block and performs the interface operations with external devices such as memory and an I/O bus interface controller. The custom processing blocks are connected to the processor by a 64-bit bus.

Table 6 shows a performance comparison of the proposed extension against other reported architectures. As the focus of this paper is to describe the KASUMI extension and the general approach of how this can be integrated into a RISC processor, the area data shown in the table includes only the area required by the extension itself, i.e. no other parts of the processor are considered. As seen from the table, the number of hardware resources required by the extension is similar to those required by the architectures that implement the KASUMI algorithm using a reuse or hybrid approach. Note that, apart from the pipelined architecture described in [10], the proposed extension achieves the highest throughput/area ratio due to the efficient reutilization of the optimized two-round ciphering datapath.

## 6. Conclusions

This paper proposed a processor-based approach to the problem of efficiently implementing the KASUMI algorithm. The general approach consisted of three phases. First, the design of a high performance hardware module that performs two rounds of the KASUMI algorithm. Second, the addition of a functional unit to a RISC processor core intended to be used in embedded environments. Third, the extension of the instruction set of the processor to exploit the capabilities of the new hardware. Replacing a long sequence of arithmetic and logical instructions by dedicated hardware reduces code size by two orders of magnitude and, consequently, the number of clock cycles needed for completion of the ciphering process. The addition of a specialized hardware module for encryption avoids requesting that service from an external coprocessor. It is important to mention that the processor used was selected to validate the approach because of its simplicity; but this general approach can be used to add a similar extension unit to other RISC-like processors.

The proposed approach scheme is a good alternative when the security functions must coexist with other operations as the functional unit for encryption does not interfere with a number of other custom modules the processor core may contain for different purposes. The extension can be used to implement other KASUMI-based algorithms, such as A5/3 and GEA3. This approach can also be adapted to implement other Feistel-like encryption algorithms.

## References

[1] 3rd Generation Partnership Program. Security Architecture. Technical Specification 33.102, Release 5, Version 5.2.0, 2003.

[2] 3rd Generation Partnership Program. Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 1: *f8* and *f9* Specification. Technical Specification 35.201, Release 5, Version 5.0.0, 2002.

[3] 3rd Generation Partnership Program. Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 2: KASUMI Specification. Technical Specification 35.202, Release 5, Version 5.0.0, 2002.

[4] N. Valtteri, K. Nyberg. UMTS Security. John Wiley & Sons Ltd, England, 2003.

[5] B. Preneel, A. Bosselaers, A. Biryukov, J. Stern, L. Granboulan, P. Nguyen, D. Pointcheval, H. Dobbertin, T. Lange, C. Paar, H. Handschuh, K. Nguyen, P. Roelse, S. Babbage, M. N¨aslund, H. Gilbert. Open problems in Cryptology. STORK Project report RUB-D6-2.1, June 2003. Available at www.stork.eu.org.

[6] M. Dworkin. Recommendation for Block Cipher Modes of Operation. NIST Special Publication 800-38A, 2001.

[7] T. Balderas, R. Cumplido. An Efficient Reuse-Based Approach to Implement the 3GPP KASUMI Block Cipher. Proceedings of the First International Conference on Electrical and Electronics Engineering and Tenth Conference on Electrical Engineering ICEEE/CIE, Acapulco, Mexico, 2004, p. 113–118.

[8] T. Balderas, R. Cumplido. "An Efficient Hardware Implementation of the KASUMI Block Cipher for Third Generation Cellular Networks. Proceedings of the Global Signal Processing Conference, GSPx 2004 , Santa Clara, CA, 2004.

[9] H. Kim, Y. Choi, M. Kim, H. Ryu. "Hardware Implementation of the 3GPP KASUMI Crypto Algorithm. Proceedings of the 2002 International Technical Conference on Circuits/Systems, Computers and Communications, ITC-CSCC-2002. Phuket, Thailand, pp. 317–320.

[10] P. Kitsos, M. D. Galanis, O. Koufopavlou. High-Speed Hardware Implementations of the KASUMI Block Cipher. Proceedings of the 2004 IEEE International Symposium on Circuit and Systems ISCAS'04 , Vancouver, Canada, 2004.

[11] K. Marinis, N. K. Moshopoulos, F. Karoubalis, K. Z. Pekmestzi. On the Hardware Implementation of the 3GPP Confidentiality and Integrity Algorithms. Proceedings of the 4th International Conference on Information Security ISC 2001, Malaga, Spain, LNCS 2200/2001, p. 248–265, Springer-Verlag, 2001.

[12] A. Satoh, S. Morioka. Small and High-Speed Hardware Architectures for the 3GPP Standard Cipher KASUMI.  Proceedings of the 5th International Conference

on Information Security ISC 2002 , Sao Paulo, Brazil, LNCS 2433/2002, p. 48–62, Springer-Verlag, 2002.

[13] Ho Won Kim, Sunggu Lee. Design and Implementation of a Private and Public Key Crypto Processor and Its Application to a Security System. IEEE Transactions on Consumer Electronics, Vol. 50, No. 1, p. 214-224, Feb. 2004.

[14] A.Wallander. A VHDL Implementation of a MIPS. Project Report, Department of Computer Science and Electrical Engineering, Lulea Tekniska Universitet, Lulea, Sweden.

[15] MIPS Technologies. MIPS SDE 5.03 Programmer's Guide, Document Number MD00310, Revision 1.67, 2004.

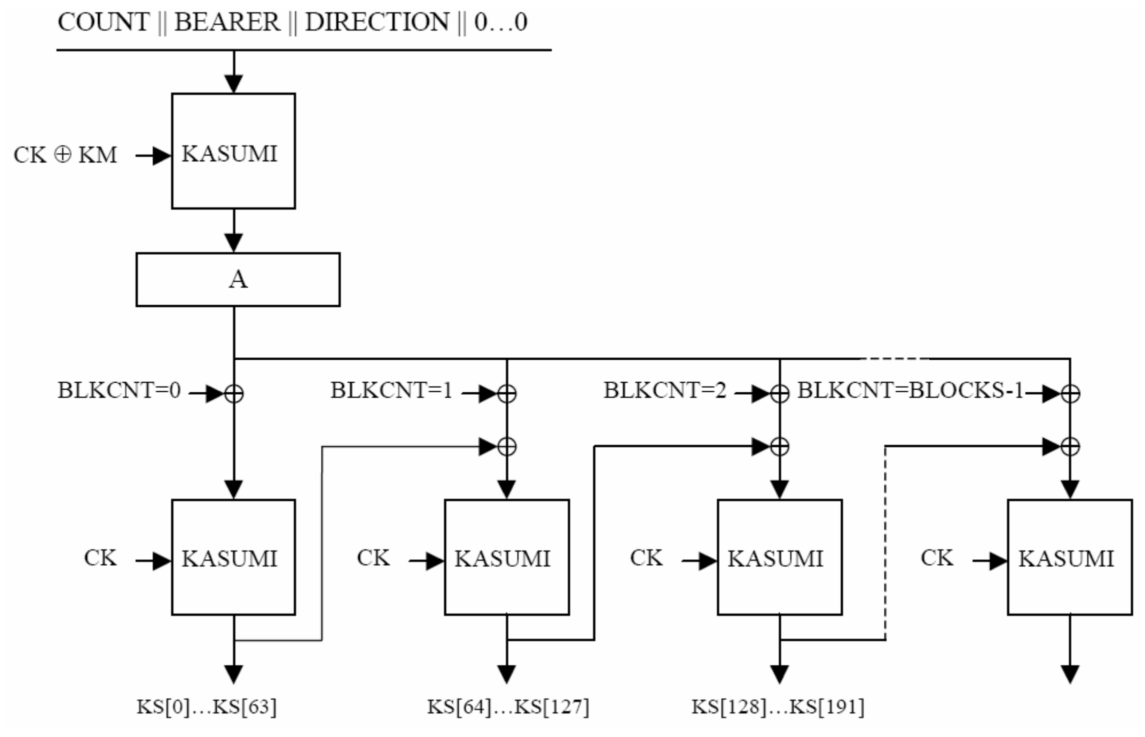


Figure 1. The *f8* confidentiality algorithm.

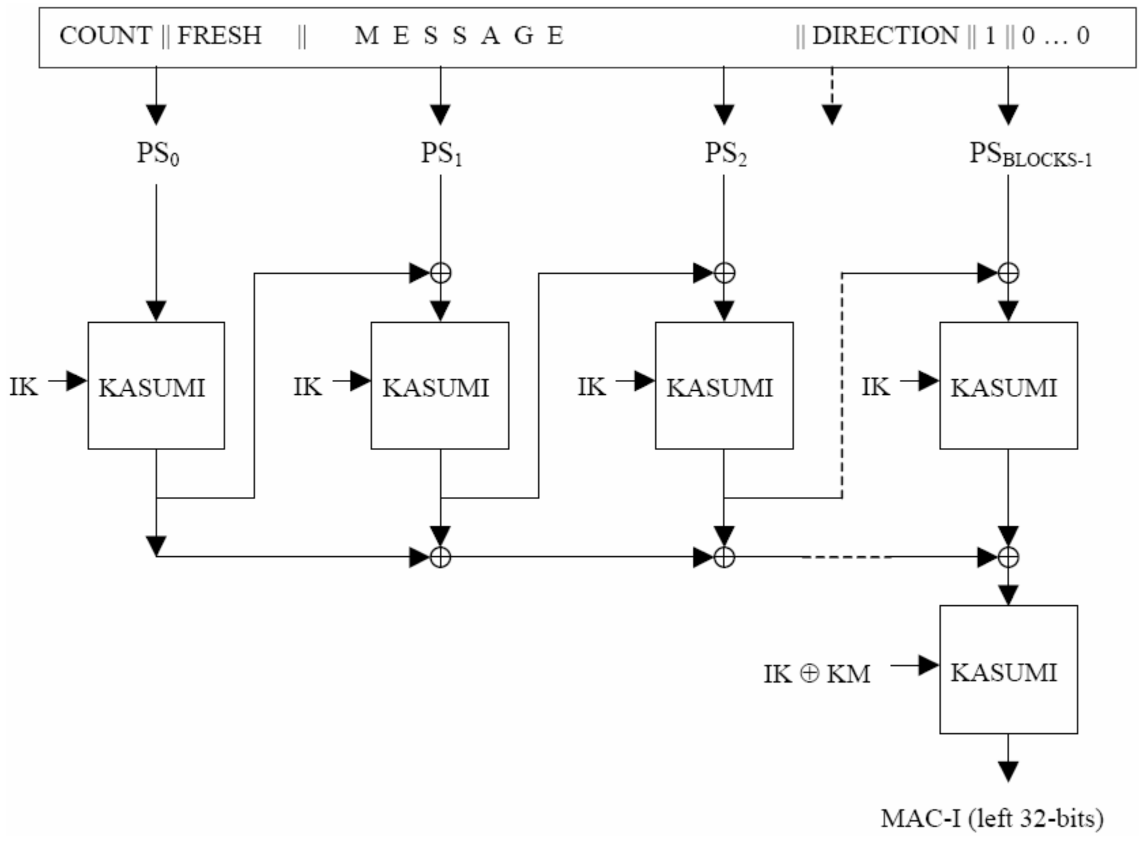


Figure 2. The *f9* integrity algorithm.

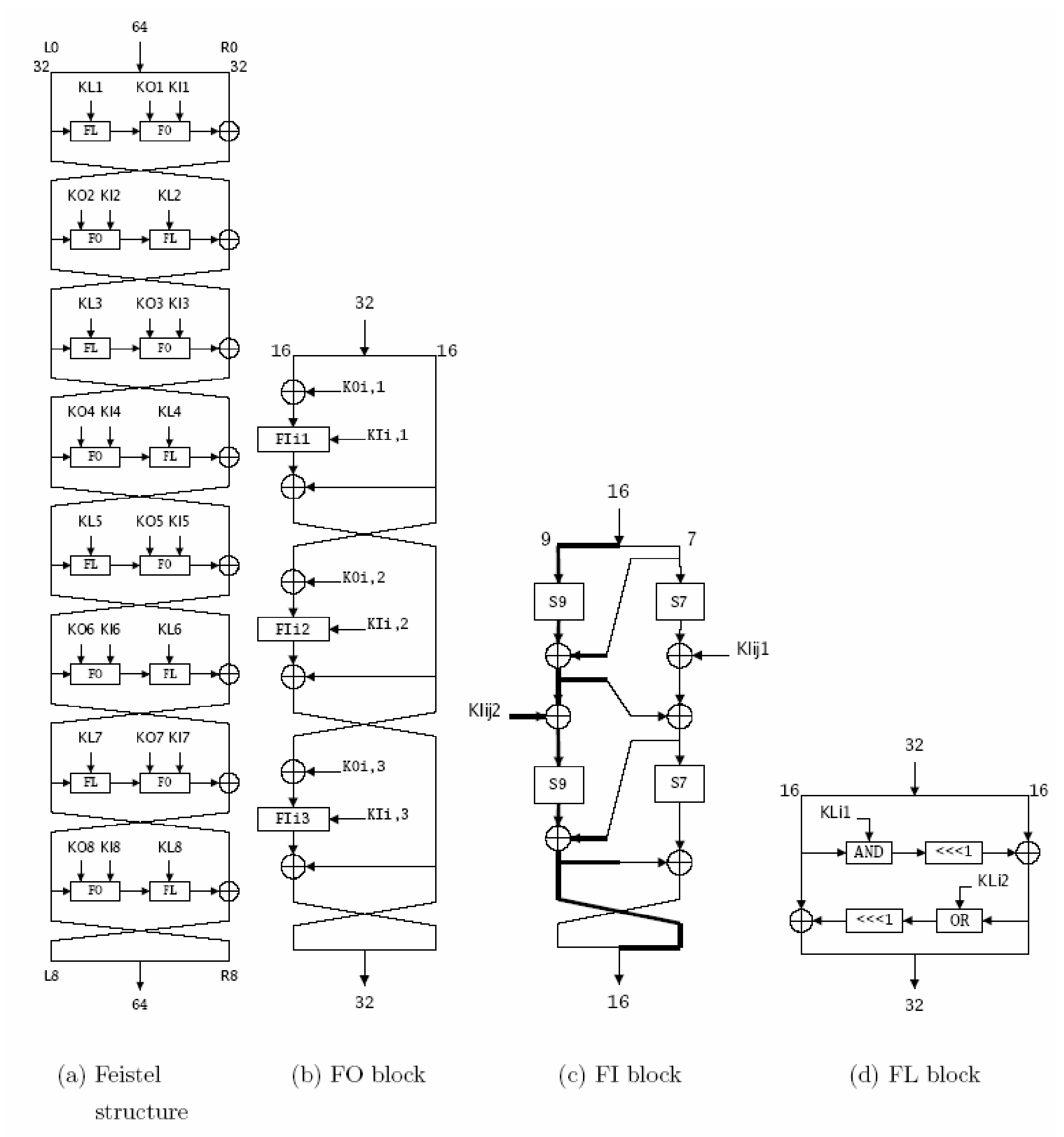


Figure 3. The KASUMI block cipher.

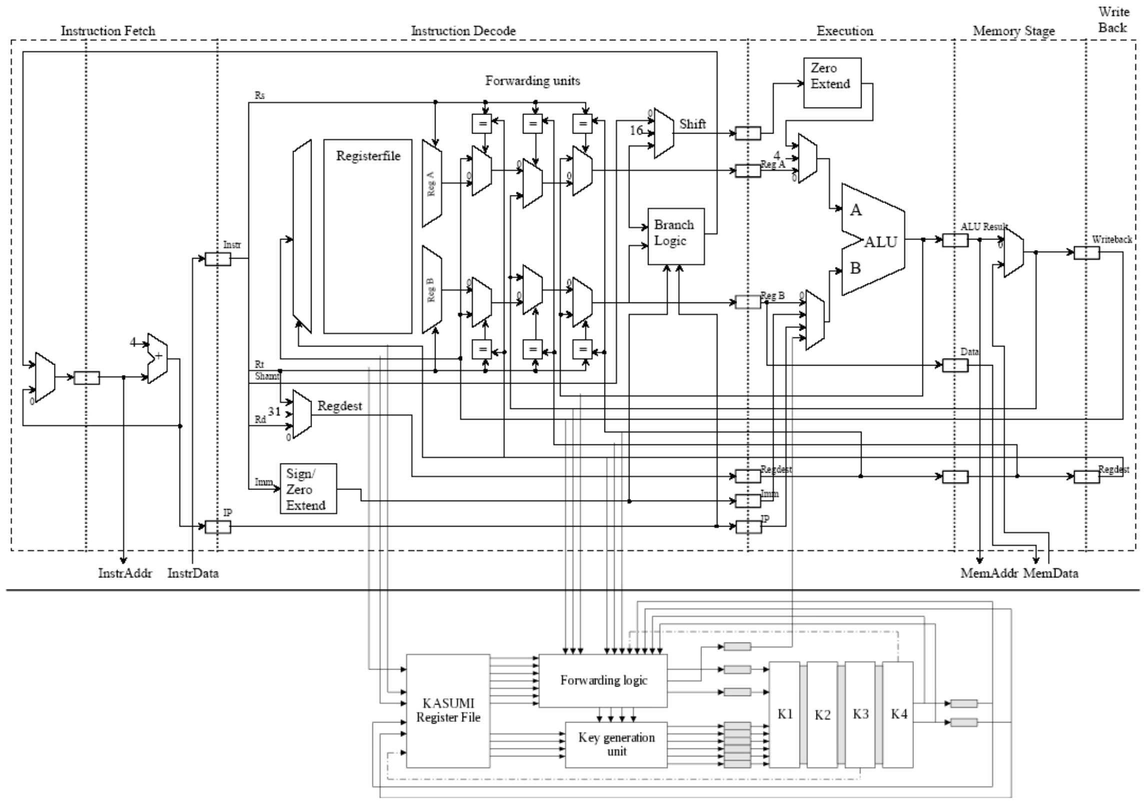


Figure 4. The MyRISC MIPS-based processor core with the extended KASUMI functional unit.

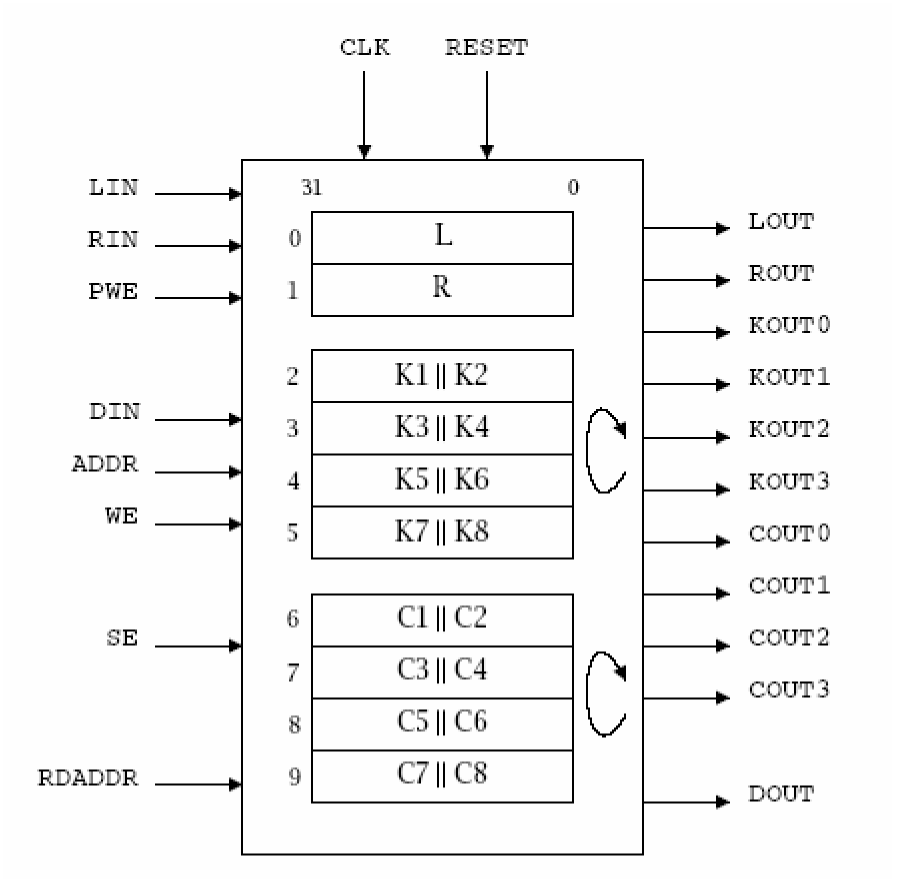


Figure 5. The organization of the extended KASUMI register file.

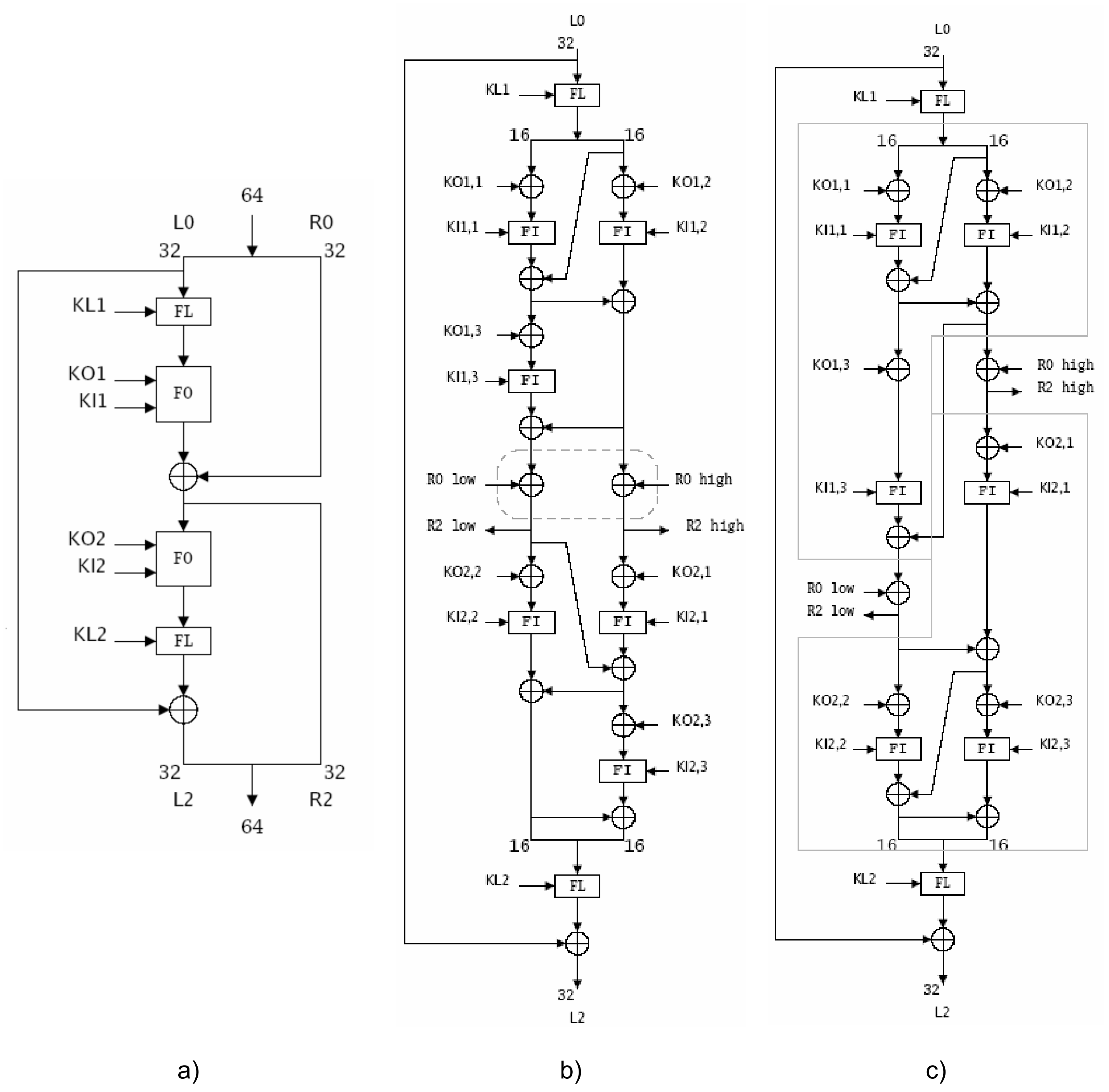


Figure 6. Sequence of steps to design a reusable datapath for two rounds of the KASUMI block cipher.

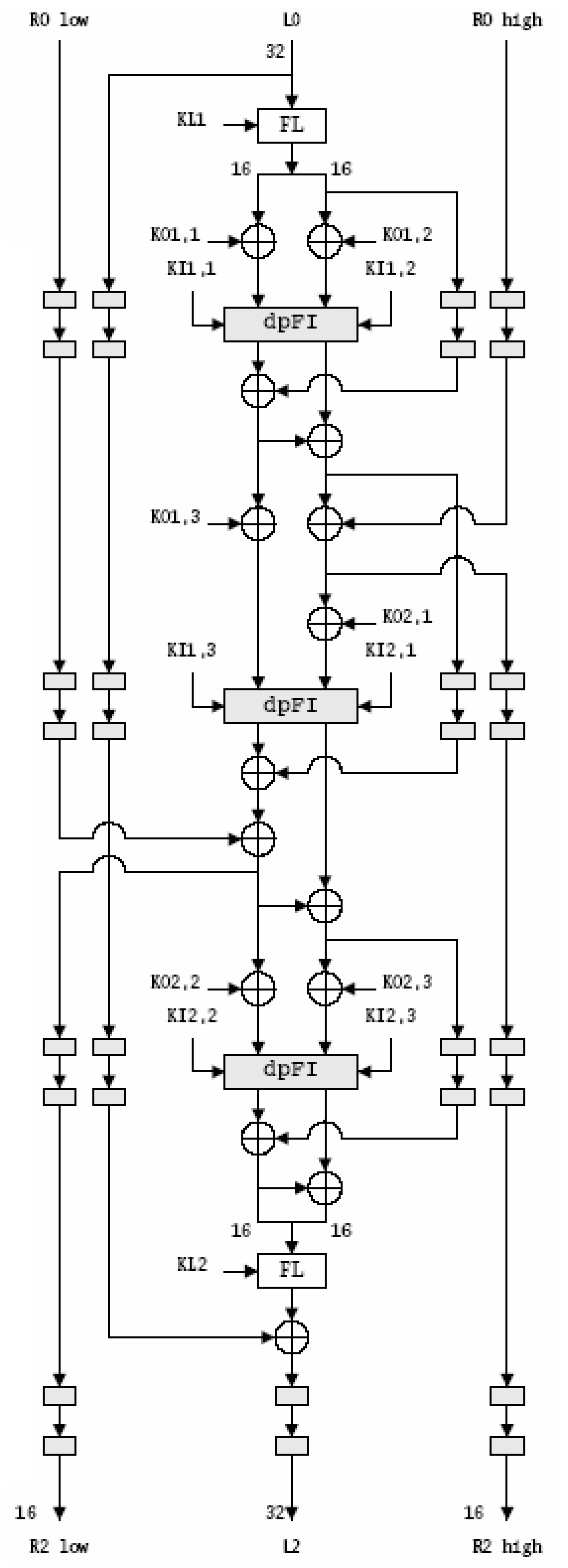


Figure 7. Pipelined datapath for the two-round sequence.

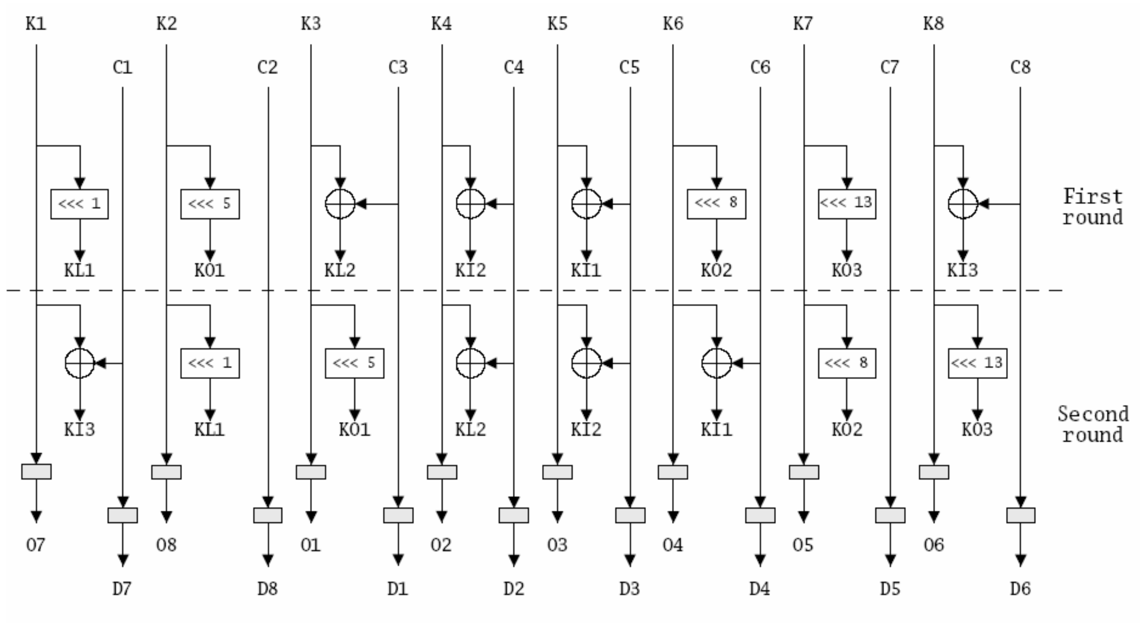


Figure 8. The Key generation unit.

| | | | | | |
|---|---|---|---|---|---|
| 000000 | Rs | Rt | KRd | X | 001010 |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |

a)

| | | | | | |
|---|---|---|---|---|---|
| 000000 | Rs | KRt | KRd | X | 001011 |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |

b)

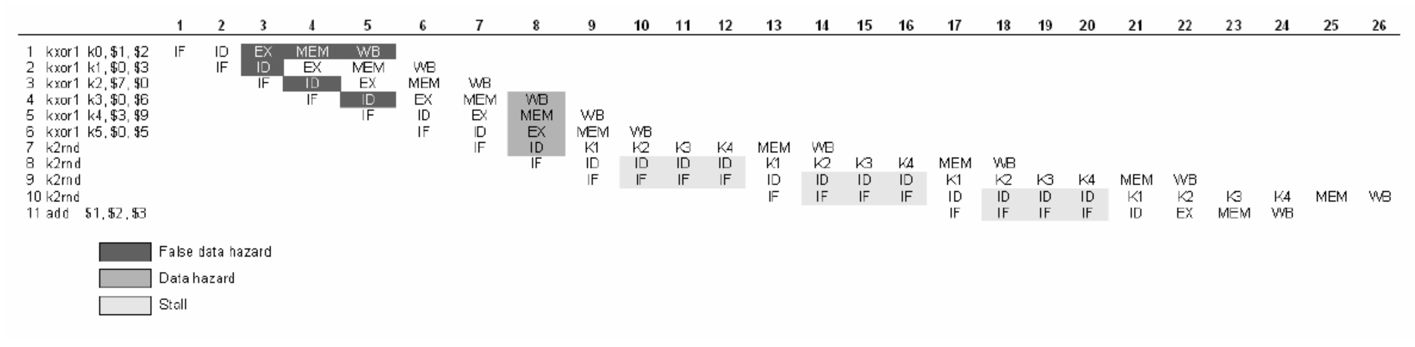
| | | | | | |
|---|---|---|---|---|---|
| 000000 | Rs | KRt | Rd | X | 110010 |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |

c)

| | | | |
|---|---|---|---|
| 101100 | X | X | X |
| 31 26 | 25 21 | 20 16 | 15 0 |

d)

Figure 9. The extended Instructions format: a) *kxor1* b) kxor2 c) kxor3 d) *k2rnd.*



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 kxor1 k0, $1, $2 | IF | ID | EX | MEM | WB | | | | | | | | | | | | | | | | | | | | | |
| 2 kxor1 k1, $0, $3 | | IF | ID | EX | MEM | WB | | | | | | | | | | | | | | | | | | | | |
| 3 kxor1 k2, $7, $0 | | | IF | ID | EX | MEM | WB | | | | | | | | | | | | | | | | | | | |
| 4 kxor1 k3, $0, $6 | | | | IF | ID | EX | MEM | WB | | | | | | | | | | | | | | | | | | |
| 5 kxor1 k4, $3, $9 | | | | | IF | ID | EX | MEM | WB | | | | | | | | | | | | | | | | | |
| 6 kxor1 k5, $0, $5 | | | | | | IF | ID | EX | MEM | WB | | | | | | | | | | | | | | | | |
| 7 k2rnd | | | | | | | IF | ID | K1 | K2 | K3 | K4 | MEM | WB | | | | | | | | | | | | |
| 8 k2rnd | | | | | | | | IF | ID | ID | ID | ID | K1 | K2 | K3 | K4 | MEM | WB | | | | | | | | |
| 9 k2rnd | | | | | | | | | IF | IF | IF | IF | ID | ID | ID | ID | K1 | K2 | K3 | K4 | MEM | WB | | | | |
| 10 k2rnd | | | | | | | | | | | | | IF | IF | IF | IF | ID | ID | ID | ID | K1 | K2 | K3 | K4 | MEM | WB |
| 11 add $1, $2, $3 | | | | | | | | | | | | | | | | | IF | IF | IF | IF | ID | EX | MEM | WB | | |

- ▇ False data hazard
- ▒ Data hazard
- ░ Stall

Figure 10. Pipelined execution of a sequence of extended instructions.

Length: 34 instructions

- Load instructions: 4
- Store instructions: 0
- Arithmetic and logic instructions: 29
- Control transfer instructions: 1

Function calls: 0

Loops: 0

If-then-else structures: 0

---

Total number of instructions executed by the function: 34

Table 1. Instruction analysis for the FI() function.

Length: 51 instructions
- Load instructions: 9
- Store instructions: 3
- Arithmetic and logic instructions: 35
- Control transfer instructions: 4

Function calls: 3
- Call 1 - FI(): 34
- Call 2 - FI(): 34
- Call 3 - FI(): 34

Loops: 0

If-then-else structures: 0

---

Total number of instructions executed by the function: 153

Table 2. Instruction analysis for the FO() function.

| | |
|---|---|
| Length: 24 instructions | |
| | Load instructions: 2 |
| | Store instructions: 0 |
| | Arithmetic and logic instructions: 21 |
| | Control transfer instructions: 1 |
| Function calls: 0 | |
| Loops: 0 | |
| If-then-else structures: 0 | |
| Total number of instructions executed by the function: 24 | |

Table 3. Instruction analysis for the FL() function.

Length: 70 instructions
- Load instructions: 14
- Store instructions: 14
- Arithmetic and logic instructions: 36
- Control transfer instructions: 6

Loops: 1
- Loop 1:
  - Iterations: 4
  - Instructions in the body of the loop: 18
  - Functions calls: 4
    - Call 1 - FL(): 24
    - Call 2 - FO(): 153
    - Call 3 - FL(): 24
    - Call 4 - FO(): 153
    - Total number for the four functions: 354
  - Length of the body of the loop including the four functions: 372
  - Total number of instructions executed by the loop: 1488

Number of instruction outside the loop: 52

---

Total number of instructions executed by the function: 1540

Table 4. Instruction analysis for the Kasumi() function.

```
Length: 138 instructions
        Load instructions: 23
        Store instructions: 21
        Arithmetic and logic instructions: 90
        Control transfer instructions: 4
Loops: 3
       Loop 1:
              Iterations: 8
              Instructions in the body of the loop: 9
              Total number of instructions executed by the loop: 72
       Loop 2:
              Iterations: 8
              Instructions in the body of the loop: 14
              Total number of instructions executed by the loop: 112
       Loop 3:
              Iterations: 8
              Instructions in the body of the loop: 88
              Total number of instructions executed by the loop: 704
       Total number of instructions executed by the three loops: 888
Number of instruction outside the loop: 27
-----------------------------------------------------------------
Total number of instructions executed by the function: 915
```

Table 5. Instruction analysis for the KeySchedule() function.

| Proposal | Approach | Clock cycles per block | Area (slices) | Frequency (MHz) | Throughput (Mbps) | Hardware efficiency (kbps/slice) |
|---|---|---|---|---|---|---|
| Works in [9] | Reuse | 8 | 650 | 20.00 | 110.00 | 169.23 |
| | Hybrid | 32 | 1100 | 33.00 | 234.00 | 212.73 |
| Works in [10] | Pipeline | 8 | 9476 | 56.00 | 3584.00 | 378.22 |
| | Hybrid | 8 | 3452 | 54.00 | 432.00 | 125.14 |
| Works in [11] | Reuse | 40 | 749 | 35.35 | 70.70 | 94.39 |
| | Pipeline | 40 | 2213 | 37.72 | 2414.08 | 1090.86 |
| Works in [12] | Reuse | 56 | 368 | 68.13 | 77.86 | 211.58 |
| | Reuse | 32 | 370 | 58.06 | 116.12 | 313.84 |
| | Reuse | 8 | 588 | 33.14 | 265.12 | 450.88 |
| Work in [13] | Hybrid | NA | 1174 | 71.00 | 568 | 483.81 |
| This work | Hybrid | 16 | 448 | 96.33 | 385.32 | 860.08 |

Table 6. Performance comparison.