Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

Reconfigurable Hardware Synthesis of the IDEA Cryptographic Algorithm

Ali E. ABDALLAH and Issam W. DAMAJ

Research Institute for Computing,
London South Bank University,
103 Borough Road,
London SE1 0AA,
United Kingdom.

`A.Abdallah@lsbu.ac.uk, I.Damaj@lsbu.ac.uk`

Abstract. The paper focuses on the synthesis of a highly parallel reconfigurable hardware implementation for the International Data Encryption Algorithm (*IDEA*). Currently, *IDEA* is well known to be a strong encryption algorithm. The use of such an algorithm within critical applications, such as military, requires efficient, highly reliable and correct hardware implementation. We will stress the affordability of such requirements by adopting a methodology that develops reconfigurable hardware circuits by following a transformational programming paradigm. The development starts from a formal functional specification stage. Then, by using function decomposition and provably correct data refinement techniques, powerful high-order functions are refined into parallel implementations described in Hoare's communicating sequential processes notation(*CSP*). The *CSP* descriptions are very closely associated with *Handle-C* hardware description language (*HDL*) program fragments. This description language is employed to target reconfigurable hardware as the final stage in the development. The targeted system in this case is the *RC-1000* reconfigurable computer. In this paper different designs for the *IDEA* corresponding to different levels of parallelism are presented. Moreover, implementation, realization, and performance analysis and evaluation are included.

1 Introduction

In the last few years, there has been dramatic advances in manufacturing Field Programmable Gate Arrays (*FPGAs*). It is now possible to make use of multi-million gates *FPGAs*. *FPGAs* offer much flexibility for the design of integrated circuits (ICs) chips for parallelism. Generally, parallelism and implementation in hardware provide us with two alternatives that can often deliver very dramatic improvements in efficiency. With the emergence of such reconfigurable hardware chips, the presence of a development environment for these scalable hardware circuits is very useful. Moreover, it would constitute the cornerstone solution for the ever-increasing need for more: efficiency, scalability and flexibility in realizing massively parallel algorithms for a wide area of applications.

The proposed rapid development model (RDM) adopts the transformational programming approach for deriving massively parallel algorithms from functional specifications [1, 2, 3]. The functional notation is used for specifying algorithms and for the reasoning about them. This is usually done by carefully combining small number of high order functions (like *map*, *zip* and *fold*) to serve as the basic building blocks for writing high-level programs. The systematic methods for massive parallelization of algorithms work by carefully composing

”off the shelf” massively parallel implementation of each of the building blocks involved in the algorithm.

To describe parallelism we follow a step-wise provably correct refinement that maps the functional specification to a network of communicating processes. Hoare’s *CSP* is used to describe the refined specification. This development step allows issues of immense practical importance (such as data distribution, network topology, and locality of communications) to be carefully reasoned about. Relating the Functional Programming and *CSP* fields gives the ability to exploit a well-established functional programming paradigms and transformation techniques in order to develop efficient *CSP* processes.

The final development stage follows the skeleton built by the previous stage, i.e. the refinement to *CSP* stage, to realize a corresponding reconfigurable hardware circuit. The reconfigurable hardware realization step is done using *Handel-C* an automated compilation development model [4]. *Handel-C* uses much of the syntax of conventional C with the addition of explicit parallelism. Handel-C relies on the parallel constructs in *CSP* to model concurrent hardware resources. Accordingly, algorithms described with *CSP* could be implemented with *Handle-C*. An overview of the transformational derivation and the hardware realization are shown in Figure 1.

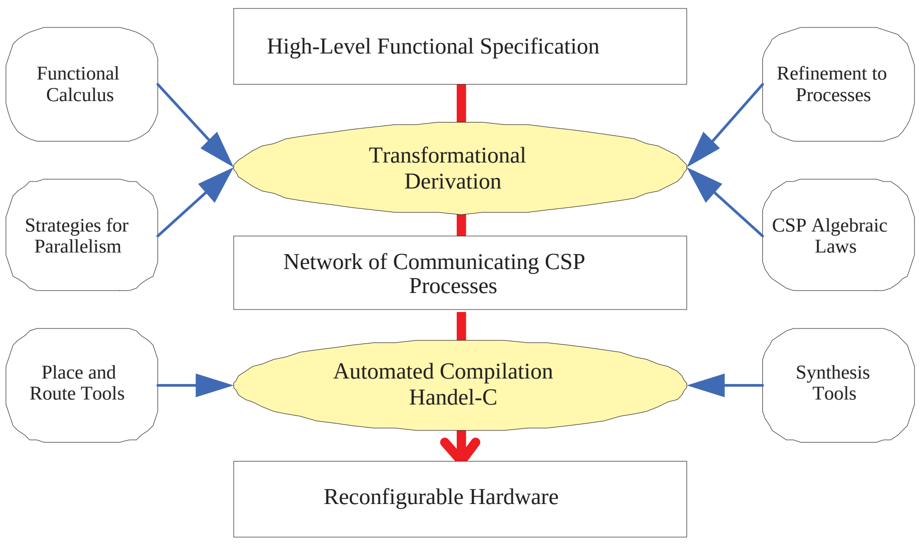
High-Level Functional Specification
Functional Calculus
Refinement to Processes
Transformational Derivation
Strategies for Parallelism
CSP Algebraic Laws
Network of Communicating CSP Processes
Place and Route Tools
Automated Compilation Handel-C
Synthesis Tools
Reconfigurable Hardware

Figure 1: An overview of the transformational derivation and the hardware realization processes.

2 Background and Previous Work

Abdallah and Hawkins defined in [2] some constructs used in the development model. This looked in some depth at data refinement; the means of expressing structures in the specification as communication behavior in the implementation.

Firstly, streams are defined as a sequence of messages on a single channel, and correspond to a sequential method for communicating a list. Streams facilitate the communication of finite sequences and require some means of signalling the end of transmission (EOT). Secondly, vectors of items are a means of communicating a list on more than one channel. The assumption is that there are as many channels in the vector as there are items in the list, such that each item is communicated on its own channel. Thirdly, vectors of streams are the parallel composition of n streams, each communicating a sublist independently as a stream. Each stream has its own end-of-transmission signal (EOT), and they can finish transmitting at different times. Lastly, streams of vectors is defined where a complete sublist is communicated in a single step.

3 Data Refinement

In the following subsections, we present some data types used for refinement.

3.1 Stream of Values

The stream is a purely sequential method of communicating a group of values. It comprises a sequence of messages on a channel, with each message representing a value. Values are communicated one after the other. Assuming the stream is finite, after the last value has been communicated, the end of transmission (EOT) on a different channel will be signaled. Given some type A, a stream containing values of type A is denoted as $\langle A \rangle$.

3.2 Vector of n Values

Each item to be communicated by the vector will be dealt with independently in parallel. A vector refinement of a simple list of items will communicate the entire structure in a single. Given some type A, a vector of length n, containing values of type A, is denoted as $\lfloor A \rfloor_n$.

3.3 Refinement of a List of Lists

Whenever dealing with multi-dimensional data structures, for example, lists of lists, implementation options arise from differing compositions of our primitive data refinements - streams and vectors. Examples of the combined forms are the *Stream of Streams*, *Streams of Vectors*, *Vectors of Streams*, and *Vectors of Vectors*. These forms are denoted by:

$$\langle S_1, S_2, ..., S_n \rangle$$
$$\langle V_1, V_2, ..., V_n \rangle$$
$$\lfloor S_1, S_2, ..., S_n \rfloor$$
$$\lfloor V_1, V_2, ..., V_n \rfloor$$

4 High-Order Functions

Functional programming environments facilitate reusability through high-order-functions. Many algorithms can be built from components which are instances of some more general scheme. In this section we introduce the refinement of some high-order-functions detailed in [2].

Map applies a function to a list of items. Thus, in the functional setting, we have:

$$map\, f\ [x_1, x_2, ..., x_n] = [f(x_1), f(x_2), ..., f(x_n)]$$

Refining to *CSP* we have:

$$VMAP_n(F) = \ ||_{i=1}^{i=n} F[in_i/in, out_i/out]$$

where, F is the refinement of f. A data parallel processes visualization of map $VMAP_n(F)$ is shown in Figure 2.

The *fold* family of functions is used to *reduce* a list by inserting a binary operator between each neighboring pair of elements. The basic fold operator (/) has no concept of direction and as such requires an associative binary operator to be well defined.

$$f\ /\ [x_1, x_2, ..., x_n] = x_1 f x_2 ... f x_n$$

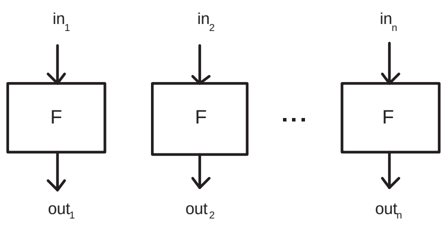

Figure 2: The Process $VMAP_n(F)$.

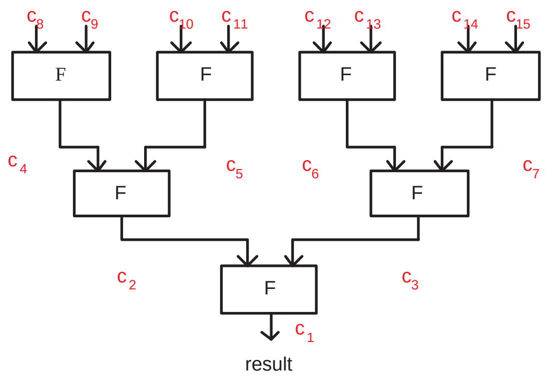

Figure 3: The Process $VFOLD_n(F)$.

Refining to *CSP* we have:

$$VFOLD_n(F) = \|_{i=1}^{i=n} F[c_i/out, c_{2i}/in_1, c_{2i+1}/in_2]$$

where, F is the refinement of the operator f. An instance of *VFOLD* is shown in Figure 3.

The high-order-function *zipWith* is used to zip two lists (taking one element from each list) with a certain operation.

$$zipWith\ f\ [x_1, x_2, ..., x_n][y_1, y_2, ..., y_n] = [x_1fy_1, x_2fy_2, ..., x_nfy_n]$$

Refining to *CSP* we have:

$$VZIP_n(F) = \|_{i=1}^{i=n} F[c_i/out, a/in_1, b/in_2]$$

5 The IDEA Algorithm

Cryptographic algorithms are an essential part in security. A well known cryptographic algorithm is the Data Encryption Standard (DES) [5, 6], widely adopted in security products. Another cryptographic algorithm is the International Data Encryption Algorithm, *IDEA* [7, 6]. Due to its high immunity to attacks [8, 6], *IDEA* is considered as one of the most important post-*DES* cryptographic algorithms.

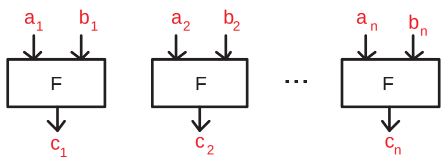

Figure 4: The Process $VZIP(F)$.

The *IDEA* algorithm is the evolution of an initial algorithm (the Proposed Encryption Standard, or *PES*) devised by Xuejia Lai and James Massey [7]. Some authors [6, 8] consider *IDEA* as one of the most secure cryptographic algorithms available at this time. In fact, there is no linear cryptanalytic attacks on *IDEA*, and there are no known algebraic weaknesses in *IDEA* other than the one discovered by Daemen [9]. Daemen discovered a weakness by using a class of 251 weak keys during encryption results in easy detection and recovery of the key. However, since there are a large number of possible keys this result has no impact on the practical security of the cipher for encryption provided, the encryption keys are chosen at random. *IDEA* is generally considered to be a very secure cipher; both the cipher development and its theoretical basis have been openly and widely discussed.

IDEA is a method to encrypt and decrypt data. A randomly secret key number is used to encrypt and decrypt the data. *IDEA* is a 64-bit iterative block cipher with a 128-bit key. The encryption process requires eight complex rounds. Decryption is carried out in the same manner as encryption once the decryption subkeys have been calculated from the encryption subkeys. The cipher structure was designed to be easily implemented in both software and hardware [10].

Hardware implementation of this cryptographic algorithm has been an active area of research. Davor and Mario presented an *FPGA* core implementation for the *IDEA*, which was addressed in [11]. They used a system with single core module to implement the *IDEA*. This module was implemented using a *Xilinx FPGA*. Cheung et al in [12] investigated a high-performance implementation of the *IDEA* using both bit-parallel and bit-serial architectures. They used a *Xilinx* Virtex XCV300-6 and XCV1000-6 *FPGAs* to evaluate and analyse the performance of the implementations. Beuchat et al in [13] presented a high-speed *FPGA* implementation of the *IDEA*. In [14] *IDEA* was addressed presenting hardware software tri-design of encryption for mobile communication units. A comparison was given between a *DSP* processor from Texas Instruments and the *Xilinx XC4000* series *FPGAs*. In [14] *VLSI* Implementation of the *IDEA* is presented. Allen et al in [15] presented an implementation comparison for the *IDEA* between the *SRC-6E* and *HC-36* general reconfigurable computers.

6 IDEA Formal Functional Specification

We view the *IDEA* algorithm as of three main blocks. A global view of these blocks would show the encryption (or decryption) as a block with 2 inputs, the private key and the plaintext (or ciphertext) and outputting the ciphertext (or plaintext). The two remaining blocks are for encryption and decryption subkeys generation. In the case of encryption subkeys generation, the block will take the private key as an input and outputs the desired subkeys. The decryption subkeys generator will input the generated encryption subkeys and output the decryption subkeys. As a first step, we define some types to be used in the following specification:

```
type Private    = [Bool]     type SubKey     = Int
type Plaintext  = [Int]      type Ciphertext = [Int]
modVal          = 65536
```

6.1 Basic Building Blocks

Three different key primitive building blocks are used within the *IDEA*:

- Bit-wise exclusive OR.
- Addition of 16-bit integers modulo 2^{16}(*modulo* 65536).

- Multiplication of 16-bit integers modulo $2^{16} + 1(\textit{modulo } 65537)$, where an all zeros input block is considered as 2^{16}.

6.2 Encryption Subkeys Generation

As shown in Figure 5, 52 16-bit subkeys are generated from the 128-bit encryption key. The algorithm for generation is as follows:

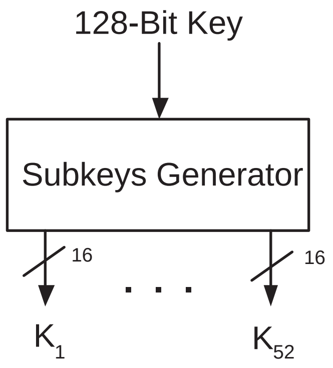

Figure 5: Subkeys Generator.

- The first eight subkeys are selected directly from the key by partitioning the key (128-bit list) into eight segments of equal length (16-bit).
- A circular shift of 25-bit positions is applied to the key of the previous step, and the eight subkeys are then extracted.
- This procedure is repeated until all 52 subkeys are generated i.e. 8-times and 4 subkeys are extracted in the final step.

In the following specification the subkeys generation is specified as the function *generateEncSubKeys*, this function takes the encryption key as input and outputs a list corresponding to the 52 16-bit subkeys. Tracing the steps of the function, it firstly takes the first eight rotations of the input key using the function *keyRotation* and generates accordingly the corresponding subkeys for each rotation through the function *generateSubKeys*. The generated subkeys are then concatenated in one list. The 52 subkeys are then extracted from the list and converted to integers equivalent to the 16-element list of *bool* representing each subkey. The conversion is done using the function *btoi*.

```
generateEncSubKeys :: Private -> [SubKey]
generateEncSubKeys key = map (btoi) (take 52
    (foldr1 (++) (map generateSubKeys (take 8 (keyRotation key)))))
```

All the rotated keys are determined by the function *keyRotation* which repeatedly generates the rotated keys. This function uses the polymorphic function *repeated* which takes a function *f* and a list *xs* and repeatedly applies the function *f* to *xs*. In this case, it repeatedly rotates the key in 25-bits steps. The rotation values would be $0, 25, 50, 75, 100, 125, 22, 47$ from the original key position.

```
keyRotation :: Private -> [[Bool]]
keyRotation key = take 8 (repeated (shift 25) key)

repeated :: (a -> a) -> a -> [a]
repeated f x = x: repeated f (f x)

shift :: Int -> [a] -> [a]
shift n key = (drop n key) ++ (take n key)
```

To generate the 16-bit subkeys from the rotated keys, the high-order function *map* is applied in the function *generateEncSubKeys* to the function *generateSubKeys* over the list of rotated keys. The function *generateSubKeys* employs *segs*, which selects *n* sublists from a list *xs*:

```
generateSubKeys :: Private -> [SubKey]
generateSubKeys key = segs 16 key

segs :: Int -> [a] -> [[a]]
segs n [] = []
segs n xs = (take n xs) : segs n (drop n xs)
```

We have the following assertion holding for all lists *xs*:

$$++/(segs\ n\ xs) = xs$$

Finally, the desired subkeys are packed in lists of 6 elements in one list of lists using the function *pack*.

```
pack :: [a] -> [[a]]
pack = segs 6
```

6.3 Decryption Subkeys Generation

After specifying the encryption subkeys generation, now we can introduce the decryption subkeys generation, where, every decryption subkey is a function of one of the encryption subkeys. The relation between the encryption and the decryption subkeys is as specified in the function *generateDecSubKeys*. This function is done by mapping a function *perform* to a prepared list of indices. The preparation of the indices list *indices* is done as shown in Figure 6. Furthermore, the function *perform* employs *addInv* and *mulInv*, which correspond to the additive and multiplicative inverse respectively. This function also uses the high-order function *mapWith* that takes a list of functions and a list of values and applies (using the function *apply*) each function in the first list to the corresponding value in the second list (using the high-order-function *zipWith*).

```
generateDecSubKeys :: [SubKey] -> [SubKey]
generateDecSubKeys eKeys  = take 52 (foldr1 (++) (map perform indices))
  where
    indices = mapWith fs (map reverse (pack (reverse [l | l<-[0..51]])))
    f1(xs) = shift 2 xs
    f2(xs) = zipWith (+) (copy (xs!!2) 6) [0, 2, 1, 3, -2, -1]
    f3 = id
    fs = [f1, f2, f2, f2, f2, f2, f2, f2, f3]
    perform(as) = mapWith [mulInv , addInv, addInv, mulInv, id, id]
                          (zipWith (!!) (copy eKeys 6) as)
copy :: a -> Int -> [a]
copy x n = [x | i <- [1..n]]
```

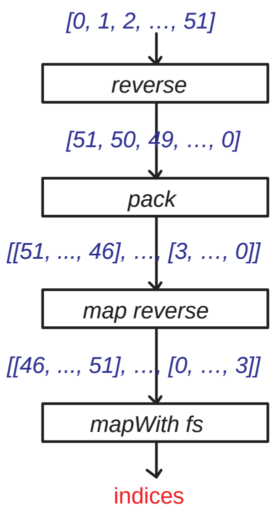

Figure 6: Indices permutation.

```
mapWith ::  [(a -> b)] -> [a] -> [b]
mapWith fs = zipWith (apply) fs

apply :: (a -> b) -> a -> b
apply f = f
```

Moving to the formal specification of modular arithmetic operations employed in the *IDEA* decryption. The additive inverse *(modulo* 2^{16}*)* and the multiplicative inverse *(modulo* $2^{16}+1$*)*. We specify these operations as the functions *addInv* and *mulInv*. The function *addInv* is simply the input number subtracted from the modulus value:

```
addInv :: Int -> Int
addInv a = modVal - a
```

To calculate the multiplicative inverse, the Extended Euclidean algorithm [16] is used, The steps to calculate the multiplicative inverse are clarified in Figure 7. Accordingly, the functional specification is as follows:

```
mulInv :: Int -> Int
mulInv 0 = 0
mulInv b = if (y < 0) then ((modVal +1) + y) else (y)
  where
    y = (extendedEucA (modVal +1) b)!!2

extendedEucA :: Int -> Int -> [Int]
extendedEucA a b
| b == 0 = [a, 1, 0]
| otherwise = iterateSteps [a, b, 0, 1, 1, 0]

iterateSteps ls = if ((ls[1]) > 0)
                  then (iterateSteps s2)
                  else ([(ls[0]), (ls[3]), (ls[5])])
  where
    s1 = (step1 ls)
    s2 = (step2 [(ls[1]), (s1[1]), (ls[2]), (s1[2]), (ls[4]), (s1[3])])
```

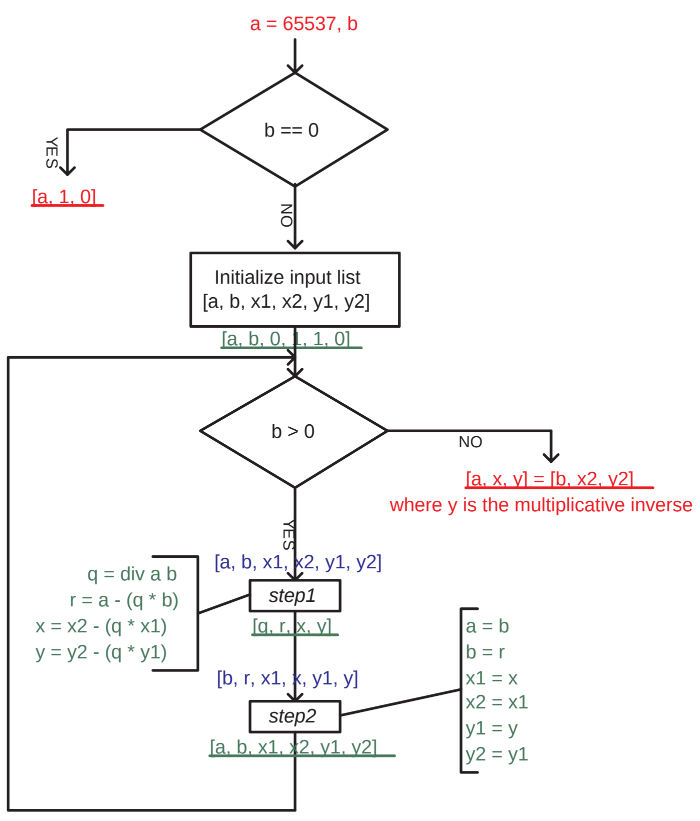

Figure 7: Extended Euclidean algorithm steps flow chart.

```
step1 :: [Int] -> [Int]
step1 ls1 = [q ,
            (ls1[0]) - (q * (ls1[1])),
            (ls1[3]) - (q * (ls1[2])),
            (ls1[5]) - (q * (ls1[4]))]
  where
    q = div (ls1[0]) (ls1[1])

step2 :: [Int] -> [Int]
step2 ls1 = [(ls1[0]), (ls1[1]), (ls1[3]), (ls1[2]), (ls1[5]), (ls1[4])]
```

6.4 IDEA Encryption and Decryption

The encryption (decryption) subkeys are made ready for the encryption (decryption) using the specified functions *generateEncSubKeys* and *generateDecSubKeys*. The encryption (decryption) works by taking a list of elements representing the plaintext (ciphertext) and the private key. Then, the list of plaintext (ciphertext) is segmented as segments of 4-elements each element representing a 16-bit word. These packed lists are then passed to encryption or decryption along with the input private key. A functional specification of *IDEA* encryption is formulated as a function *encryption*. The encryption function works by firstly segmenting the input list using the function *segs*. Secondly, it maps the function responsible for a single block encryption with the input private key to all segmented input list elements. The function responsible for encrypting a single 4-element list is called *encryptSegs*.

```
encryption :: Private -> Plaintext -> Ciphertext
encryption key ls = concat (map (encryptSegs key) (segs 4 ls))
```

A different specification that considers the input plaintext as an already segmented list *ls*:

```
encryption :: Private -> [Plaintext] -> [Ciphertext]
encryption key ls = map (encryptSegs key) ls
```

The decryption has a similar specification. Figure 8 shows the structure and the block diagram for the *IDEA*. A single 64-bit block from the plaintext segmented as a list of 4 elements each of 16-bit inputs to this structure. The output has a similar type, but it represents a block from the ciphertext.

We specify the encryption of one block as the function *encryptSegs*. This function firstly packs the encryption subkeys. Then, it folds (using the high-order-function *foldl*) with an initial list *xs* the function *singleRound* distributing the packed subkeys to each round. Note that the function *singleRound* is the formal specification of a round. The folded output is then passed to the function *outputTransformation* along with the last pack of subkeys, giving the final output. The function *outputTransformation* specifies the output transformation stage found as the final stage in *IDEA* encryption (decryption).

```
encryptSegs :: Private -> [Int] -> [Int]
encryptSegs key xs = [e, g, h, f]
  where
    kss = pack (generateEncSubKeys key)
    [a, b, c, d] = foldl singleRound xs (init kss)
    ([e, f], [g, h]) = outputTransformation  [a, c, b, d] (last kss)
```

The decryption could be specified in a similar manner.

6.4.1 Single Round Specification

The main part of the *IDEA* algorithm consists of the application of 8 similar rounds to the input plaintext and the key as shown in Figure 8. In this section we introduce the round construct by introducing each of its building blocks.

A round is specified as a function *singleRound* with two input lists, one representing the input block from the plaintext and the other a pack of subkeys. A *singleRound* works by composing three different functions *firstSubRound*, *secondSubRound*, and *thirdSubRound* (See Figure 9).

```
singleRound :: [Int] -> [Int] -> [Int]
singleRound xs ks = thirdSubRound (secondSubRound (firstSubRound ks xs))
```

The function *firstSubRound* employs modular multiplication and addition to the first 4 elements of both input lists. This function also forwards the last two subkeys from input to output list.

```
firstSubRound  :: [Int] -> [Int] -> [Int]
firstSubRound  [k1, k2, k3, k4, k5, k6] [x1,x2,x3,x4] =
    [(mulMod x1 k1), (addMod x2 k2),
    (addMod x3 k3), (mulMod x4 k4), k5, k6]
```

In *IDEA*, each plaintext bit influence every ciphertext bit. The spreading out of a single plaintext bit over many ciphertext bits hides the statistical nature of the plaintext [10]. This diffusion is provided by the basic building block of the algorithm known as the multiplication/addition (*MA*) structure shown in Figure 9. The function that specifies this structure is called *mA*, and the multiplication/addition is done using the functions *mulMod/addMod*.

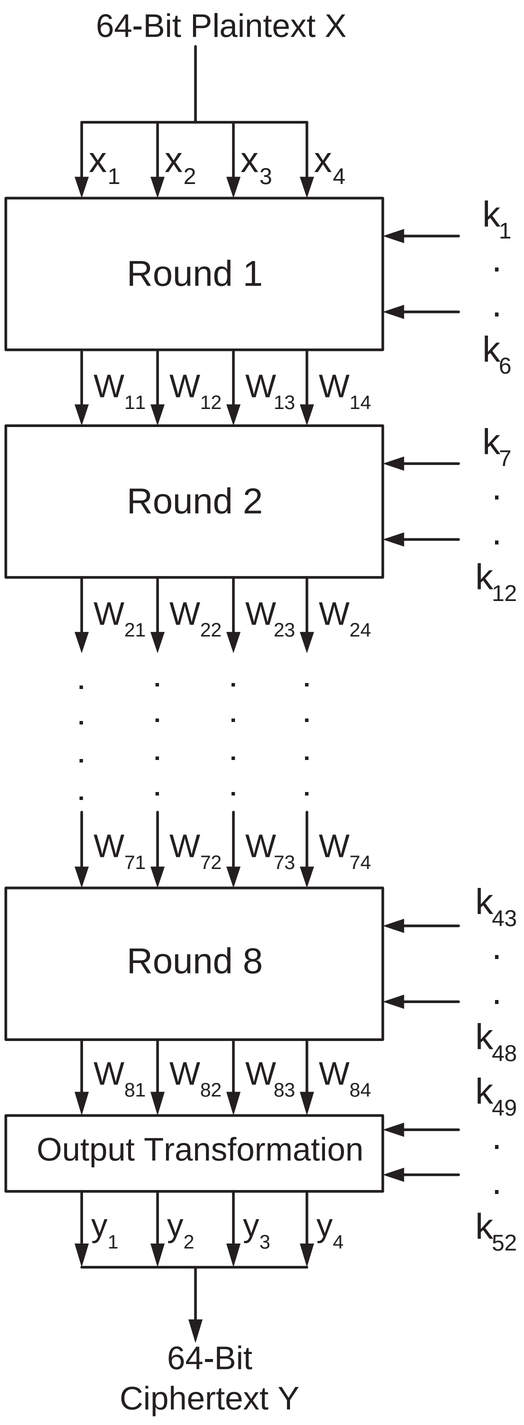

Figure 8: IDEA general structure.

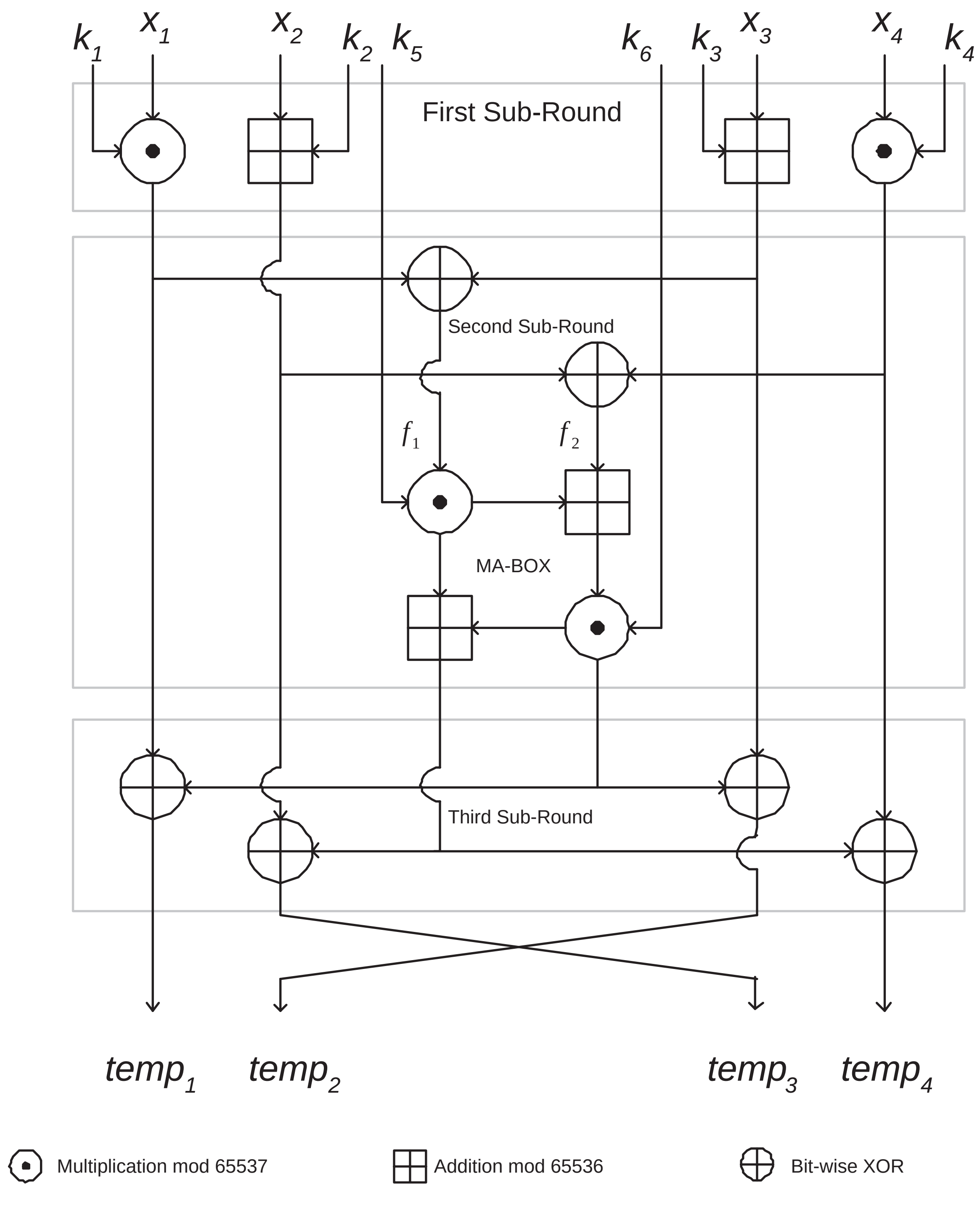

Figure 9: IDEA round.

```
mA :: [Int] -> (Int, Int)
mA [u, w, k5, k6] = (addMod a b, b)
  where
    a = mulMod u k5
    b = mulMod (addMod w a) k6
```

Thereby, the function *secondSubRound* employs the function *mA* over two subkeys and the result of XORing 4 elements from its own input.

```
secondSubRound  :: [Int] -> [Int]
secondSubRound [v1, v2, v3, v4, k5, k6] = [v1, v2, v3, v4, q1, q2]
  where
    (p1, p2) = ((fullexor v1 v3), (fullexor v2 v4))
    (q1, q2) = mA [p1, p2, k5, k6]
```

A third subround is specified to complete the scene of a whole round. This function, namely *thirdSubRound*, is responsible for XORing its inputs. For this sake the high-order-function *zipWith* is used.

```
thirdSubRound :: [Int] -> [Int]
thirdSubRound [y1, y2, y3, y4, p1, p2] =
    zipWith fullexor [y1, y3, y2, y4] [p2, p2, p1, p1]
```

As employed in specifying a single round's constructs, the modular addition *(modulo 65536)* is specified as a function *addMod* with two inputs and one output of type *Int*. The specification can use the modulo operation *mod* to calculate the modular addition as follows:

```
addMod :: Int -> Int -> Int
addMod i1 i2 = mod (i1 + i2) modVal
```

Escaping the cost of the parallel implementation of the operation *mod* as to be implemented by hardware the functional specification is done as follows:

```
addMod :: Int -> Int -> Int
addMod i1 i2 = fullAND (i1 + i2) (modVal - 1)
```

Where, the function *fullAND* is the bit-wise logic AND. It is worth to note at this step that an iterative addition/subtraction dependant version of the operation *mod* could be done as follows:

```
mymod :: Int -> Int -> Int
mymod a b
  | a < b = a
  | otherwise = mymod (a - b) b
```

However, The significance of different versions of these operations is to be more transparent at the realization step. The modular multiplication *(modulo 65537)* is considered one of the most expensive operations used in the *IDEA* from hardware usage and/or throughput points of evaluation.

Considerable research has been done trying to afford different economical and/or efficient implementations. In [16] different designs where addressed discussing the mathematical foundation of each and giving their reconfigurable hardware implementations. An efficient implementation is suggested in [17]. We specify this algorithm as follows:

```
mulMod ::  Int -> Int ->Int mulMod x y =
    if ((mulModEfficient x y) == modVal)
    then (0)
    else (mulModEfficient x y)

mulModEfficient ::  Int -> Int -> Int mulModEfficient x y
  | (x == 0) && (y == 0) = 1
  | (x == 0) && (y /= 0) = ((modVal+1) - y)
  | (x /= 0) && (y == 0) = ((modVal+1) - x)
  | otherwise = if (cL < cH)
                then (cL - cH + (modVal+1))
                else (cL - cH)
  where
    cL= b2i  (take 16 (i2b (x * y)))
    cH= b2i  (drop 16 (padWithFalse32BitR (i2b (x * y))))
```

6.4.2 Output Transformation Specification

This stage is designed to allow the decryption to have the same structure as encryption. The specification is the same as that for the function *firstSubRound*.

7 Refinement of the IDEA Formal Specification

Narrowing the distance from a specific hardware implementation, we apply the step-wise refinement suggested by the development methodology. Data and process refinements are executed with a main concern of demonstrating the design flexibility granted by the proposed methodology. Designs varying from data-parallel to pipelined are shown giving the *CSP* implementation of each.

7.1 Encryption Subkeys Generation

The following design is the refinement of the subkeys generating functions. Datatype refinement considers the input as a 128-bit integer *(Int128)* item to correspond to the 128-bit list of *bool*. An alternative implementation as a 128-element vector of *bool* items could be followed. The first implementation is chosen since, an integer as viewed in hardware is an array of individually manipulatable bits. This is nearly identical to the latter implementation, with a difference that the 128-bit integer item will be communicated on a single channel instead of a vector of channels. The output is refined to a vector of items thereat the 52 subkeys are to be taken. Generally, this design will fork a parallel computation aiming for an expectedly fast subkeys generation. This is done by executing 8 parallel instances of a subkey generator leading to a parallel production of all subkeys. The first step is done by refining the function *generateEncSubKeys* as the process *GENCSKEYS,* where:

$$\begin{aligned}&\mathit{generateEncSubKeys} :: \mathit{Int}128 \to \lfloor \mathit{Int}16 \rfloor_{52}\\&\mathit{generateEncSubKeys} \sqsubseteq \mathit{GENCSKEYS}\end{aligned}$$

The *CSP* implementation that corresponds to *generateEncSubKeys* is as follows:

$$\begin{aligned}&\mathit{GENCSKEYS} =\\&\quad (\mathit{in}?\mathit{key} \to \mathit{SKIP});\ \mathit{KEYROTATION}(\mathit{key}) >>_8 \mathit{VMAP}_8(\mathit{GSUBKEYS}) >>_8 \mathit{CONCAT}\end{aligned}$$

While, the following holds:

$$\begin{aligned} \mathit{keyRotation} &\sqsubseteq \mathit{KEYROTATION} \\ \mathit{generateSubKeys} &\sqsubseteq \mathit{GSUBKEYS} \\ \mathit{concat} &\sqsubseteq \mathit{CONCAT} \end{aligned}$$

Figure 10 is a visualization of the process *GENCSKEYS*.

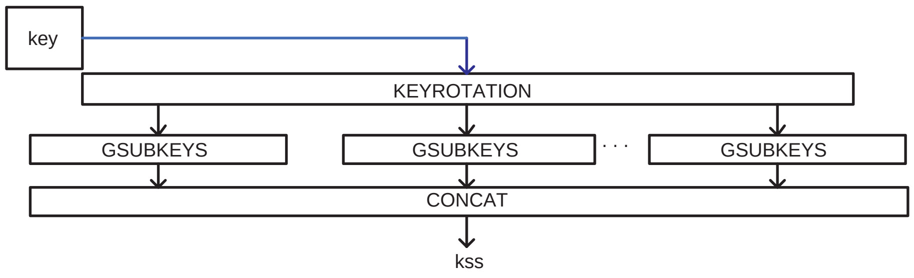

Figure 10: The process *GENCSKEYS*.

Moving to the sub-blocks of this generator, the functional specification for the function *keyRotation* is:

```
keyRotation :: Private -> [[Bool]]
keyRotation key = take 8 (repeated (shift 25) key)
```

Where, *take 8 (repeated (shift 25) key)* could be rewritten, depending on the specification of *repeated*, as:

```
take 8 (repeated (shift 25) key)=
    map (flip shift key) [0, 25, 50, 75, 100, 125, 22, 47]
```

The final specification would be:

```
keyRotation :: Private -> [[Bool]]
keyRotation key = map (btoi) (map (flip shift key) ls)
  where
    ls = [0, 25, 50, 75, 100, 125, 22, 47]
```

In this design we considered the refinement of the input key type to be an item, while the list *ls* and the output rotated keys as vectors of items.

$$\mathit{keyRotation} :: \mathit{Int}128 \rightarrow \lfloor \mathit{Int}128 \rfloor_8$$

The *CSP* implementation of the functional specification refines *keyRotation* to a process *KEYROTATION*.

The key could be passed as an argument to each of the processes *SHIFT(key)*, while distributing the list *ls* elements to the parallel processes as shown in Figure 11. The is described as follows:

$$\mathit{KEYROTATION}(\mathit{key}) = \mathit{VMAP}_8(\mathit{PRD}(\mathit{key}) \triangleright \mathit{SHIFT})$$

The key could be explicitly passed to the process *SHIFT*. The effect of applying this step to the previous design can be visualized as in Figure 12. In the above version the key is locally produced and fed to each process *SHIFT*. The effect of having 8 parallel copies of *PRD(key)* communicating with 8 instances of *SHIFT* can be eliminated by factorizing the

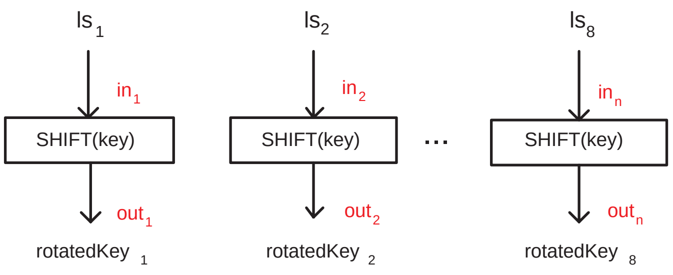

Figure 11: The process *KEYROTATION*.

process *PRD(key)* and broadcasting its output to the relevant processing elements in the network. Applying this rule will result in a semantically equivalent version of *KEYROTATION* which has a different layout. This is shown in Figure 13. The formal rule that justifies the above transformation is:

$$KEYROTATION(key) = BROADCAST_8(key)[d/out] \rhd_8 VMAP_8(SHIFT)$$

where,

$$shift \sqsubseteq SHIFT$$

The refinement of the input *n* and the output rotated key realises them as items.

$$shift :: Int128 \longrightarrow Int \longrightarrow Int128$$

The *CSP* implementation of *shift* is the process *SHIFT*:

$$SHIFT = (in_1?key \rightarrow SKIP \; ||| \; in_2?n \rightarrow SKIP); \; out!(key[n..127]{+\!\!+}key[0..n])$$

where, $key[n..127]{+\!\!+}key[0..n]$ is the integer equivalent of the concatenation of the upper $127 - n$ bits of the 128-bit key, and lower n bits. Figure 14 gives a general visualization of the process *GENCSKEYS*. The next step is presenting the refinement of *generateSubKeys*. The corresponding *CSP* process is *GSUBKEYS*.

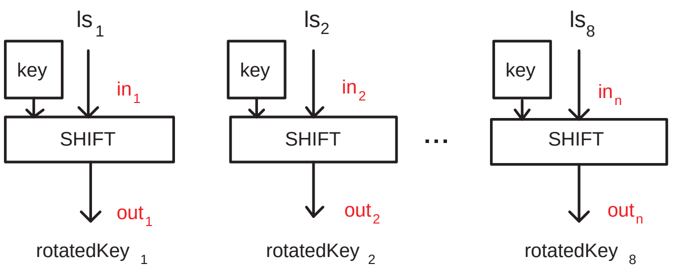

Figure 12: The process *KEYROTATION*, an alternative.

The *CSP* refinement realises the input as an item, and the output as a vector of items where each item is a list.

$$generateSubKeys :: Int128 \longrightarrow \lfloor Int16 \rfloor_8$$
$$GSUBKEYS = (in?key \rightarrow SKIP); \; SEGS(key)$$

The recursion in *segs* is unrolled for n equals 16 in a similar way to that done for *keyRotation*.

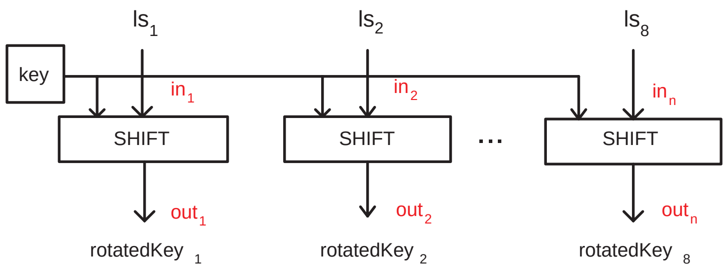

Figure 13: The process *KEYROTATION*, optimised implementation.

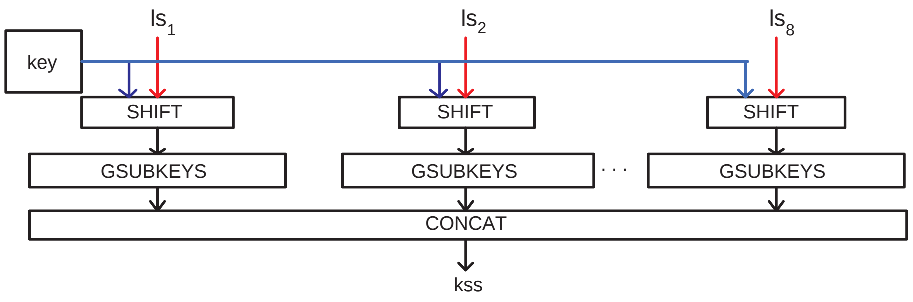

Figure 14: The Process *GENCSKEYS*.

```
segs 16 key =
    [(take 16 key), (take 16 (drop 16 key)),
    (take 16 (drop 32 key))), ..., (take 16
    (drop 112 key)))]
```

Then, the refinement to *CSP* is as follows:

$$SEGS = (in_1?n \rightarrow SKIP \mid\mid\mid in_2?key \rightarrow SKIP);$$
$$\mid\mid\mid_{i=0}^{i=\frac{length(key)}{n}-1} out[i]!(key[i*n..((i+1)*n)-1])$$

where $key[0..15], key[16..31], \ldots$ are the integers equivalent to each 16-bit word. The refinement of the function *pack* is the process *PACK*.

7.2 Decryption Subkeys Generation

The decryption subkeys generation is refined in two ways controlling the number of used processes. The first design replicates the use of the processes *MULINV* and *ADDINV* (the refinement of *addInv* and *mulInv*), where all subkeys are produced in parallel as a vector of vector of items. Each item is communicating on a different channel. The second design implements 4 parallel processes that are inputting the encryption subkeys as a stream of vectors and outputting the desired decryption subkeys as a stream of vectors. In the second design the replication of *MULINV* and *ADDINV* is restricted to 2 of each, thus an economical use of hardware resources is expected in the realization step.

7.2.1 Decryption Subkeys Generation - First Design

We firstly recall the part of the specification responsible for creating the permutation indices. The list *indices* is created as follows:

```
indices = mapWith fs (map reverse (pack (reverse [l |l<-[0..51]])))
f1(xs) = shift 2 xs
f2(xs) = zipWith (+) (copy (xs!!2) 6) [0, 2, 1, 3, -2, -1]
f3 = id
fs = [f1, f2, f2, f2, f2, f2, f2, f2, f3]
```

The generated list indices has the following values:

```
indices = [[48,49,50,51,46,47],
           [42,44,43,45,40,41],
           [36,38,37,39,34,35],
           [30,32,31,33,28,29],
           [24,26,25,27,22,23],
           [18,20,19,21,16,17],
           [12,14,13,15,10,11],
           [6, 8, 7, 9, 4, 5 ],
           [0, 1, 2, 3]]
```

For simplicity in implementation we replace the computational constructs with a table of values containing the required indices. Accordingly, this permutation is applied to the input encryption subkeys. The modified specification is as follows:

```
generateDecSubKeys :: [SubKey] -> [[SubKey]]
generateDecSubKeys eKeys  = map perform eKeysPerm
  where
    indices =  [[48,49,50,51,46,47],
                [42,44,43,45,40,41],
                [36,38,37,39,34,35],
                [30,32,31,33,28,29],
                [24,26,25,27,22,23],
                [18,20,19,21,16,17],
                [12,14,13,15,10,11],
                [6, 8, 7, 9, 4, 5 ],
                [0, 1, 2, 3]]
    eKeysPerm = map (zipWith (!!) (copy eKeys 6)) indices
    perform(as) = mapWith [mulInv , addInv, addInv, mulInv, id,id] as
```

The input and output to and from the process *GDSKEYS*, the refinement of *generateDecSubKeys*, are communicated as a vector and a vector of vectors:

$$\mathit{generateDecSubKeys} :: \lfloor \mathit{Int}16 \rfloor_{52} \rightarrow \lfloor \lfloor \mathit{Int}16 \rfloor_{6} \rfloor_{9}$$

The input encryption subkeys are firstly permutated according to the given *indices*, and then produced to parallel instances of the process *PERFORM* the refinement of *perform*.

$$\begin{aligned} \mathit{GDSKEYS} = {} & |||_{i=0}^{i=51}\ (\mathit{in.elements}[i]?\mathit{skeys}[i] \rightarrow \mathit{SKIP}); \\ & (\mathit{PRDp}(\mathit{skeys}) \rhd \mathit{VMAP}_9(\mathit{PERFORM})) \end{aligned}$$

$$\begin{aligned} \mathit{PERFORM} = {} & |||_{j=0}^{j=1}\ (\mathit{ADDINV}[in/in_j, out/out_j] \\ & ||| \ \mathit{MULINV}[in/in_j, out/out_j] \\ & ||| \ (\mathit{Forward}[in/in_j, out/out_j])) \end{aligned}$$

$$\mathit{PRDp}(ls) = (|||_{i \in P, j=0}^{j=53}\ \mathit{out.elements}[j]!ls[i]) \rightarrow \mathit{SKIP}$$

$$\begin{aligned}P = \{&48, 49, 50, 51, 46, 47, 42, 44, 43, 45, 40, 41,\\ &36, 38, 37, 39, 34, 35, 30, 32, 31, 33, 28, 29, 24, 26,\\ &25, 27, 22, 23, 18, 20, 19, 21, 16, 17, 12, 14, 13, 15,\\ &10, 11, 6, 8, 7, 9, 4, 5, 0, 1, 2, 3, 0, 0\}\end{aligned}$$

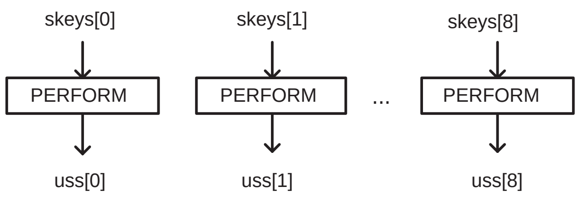

Figure 15: Decryption subkeys generation, first design.

7.2.2 Decryption Subkeys Generation - Second Design

In this design, the input and output are communicated as streams of vectors of 6 items. The input vector is ordered in the way needed for the process, where the first and the fourth elements are passed to the *MULINV* processes, the second and the third inputs are passed to the *ADDINV* processes. The last two input elements are forwarded to the output channels in their order. This process is visualized in Figure 16.

$$\mathit{generateDecSubKeys} :: \lfloor \mathit{Int}16 \rfloor_{52} \rightarrow \langle \lfloor \mathit{Int}16 \rfloor_6 \rangle$$

$$\begin{aligned}\mathit{GDSKEYS} = &\ |||_{i=0}^{i=51}\ \mathit{in.elements}[i]?\mathit{skeys}[i] \rightarrow \mathit{SKIP});\\ &(\mathit{SPRDp}(\mathit{skeys}) \rhd \mathit{SMAP}(\mathit{PERFORM})\end{aligned}$$

$$\begin{aligned}\mathit{SPRDp}(\mathit{ls}) = &((;\ )_{i=0}^{i=8}(|||_{j\in P'[i],k=0}^{k=5}\ \mathit{out.elements}[k]!\mathit{ls}[j])) \rightarrow\\ &\mathit{out.eotChannel}!\mathit{eot} \rightarrow \mathit{SKIP}\end{aligned}$$

$$\begin{aligned}P' = \{&\{48, 49, 50, 51, 46, 47\}, \{42, 44, 43, 45, 40, 41\},\\ &\{36, 38, 37, 39, 34, 35\}, \{30, 32, 31, 33, 28, 29\},\\ &\{24, 26, 25, 27, 22, 23\}, \{18, 20, 19, 21, 16, 17\},\\ &\{12, 14, 13, 15, 10, 11\}, \{6, 8, 7, 9, 4, 5\}, \{0, 1, 2, 3, 0, 0\}\}\end{aligned}$$

7.3 IDEA Encryption and Decryption

A parallel program for a block encryption (or decryption) could be viewed with different levels of parallelism. The first design is suggested to view the input subkeys as vector of vectors passed in parallel to the parallel rounds. This design replicates the process *SINGLEROUND* (which corresponds to the function *singleRound*) an 8-stage pipeline. The replication is done using a vector implementation off-the-shelf refined high-order-function *foldl*.

Another design considers input subkeys as stream of vectors using one instance of the process *SINGLEROUND*, thus a later minimal use of resources needed by *SINGLEROUND* processes. This is done using a sequential implementation of *foldl*.

A compromised design affording flexibility in controlling replication of the process *SINGLEROUND* is done by taking a part of the subkeys as vector of vectors while the remaining

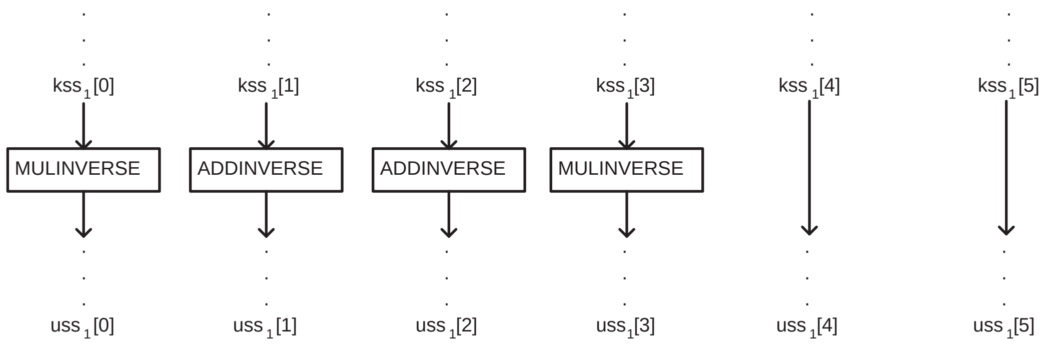

Figure 16: Decryption subkeys generation, second design.

subkeys as stream of vectors. This is a tradeoff between use of hardware resources and throughput of processing. These designs are elaborated in the following subsections.

The encryption for a whole set of plaintext also could be viewed in two levels of parallel execution. A first design could pass sequentially the plaintext blocks as a stream of vector of items. A second design could pass the blocks as streams of vector of vectors of items, replicating the whole *IDEA* block leading to a multi-way encryption. These two versions are presented for the encryption taking into consideration that the decryption is implemented similarly. Refining the input plaintext segmented blocks to a stream of vector of items, while the key is input once as an item, we get:

$$encryption :: \langle \lfloor Int16 \rfloor_4 \rangle \rightarrow \langle \lfloor Int16 \rfloor_4 \rangle$$

where:

$$ENCRYPTION(key) = SMAP(ENCRYPTSEGS(key))$$

The second version is implemented as follows, where *n* is limited to the available resources.

$$encryption :: \langle \lfloor \langle \lfloor Int16 \rfloor_4 \rangle \rfloor_n \rangle \rightarrow \langle \lfloor \langle \lfloor Int16 \rfloor_4 \rangle \rfloor_n \rangle$$
$$ENCRYPTION(key) = SMAP(VMAP_n(SMAP(ENCRYPTSEGS(key))))$$

7.3.1 IDEA First Design

The main point of the first design is to have a version of *IDEA* with all of its rounds working in parallel. This is apparent from the refinement of the input subkeys. The subkeys are refined as a vector of vectors of items and distributed to the parallel rounds. The refinement realises this function as the process *ENCRYPTSEGS* (See Figure 17), where the input and output segments are streams of vectors of items and the key is passed as an item. The key will be used to generate a vector of vectors of subkeys through *GENCSKEYS*.

$$encryptSegs :: Int128 \rightarrow \langle \lfloor Int16 \rfloor_4 \rangle \rightarrow \langle \lfloor Int16 \rfloor_4 \rangle$$
$$encryptSegs \sqsubseteq ENCRYPTSEGS$$
$$\begin{aligned} ENCRYPTSEGS = {} & GENCSKEYS \gg_{52} PACK \gg_9 \\ & (VVFOLDL(SINGLEROUND) || OTPTTRANS) \end{aligned}$$

The process *VVFOLDL(SINGLEROUND)* is an off-the-shelf refinement for the high-order-function *foldl* over a vector of vectors of items.

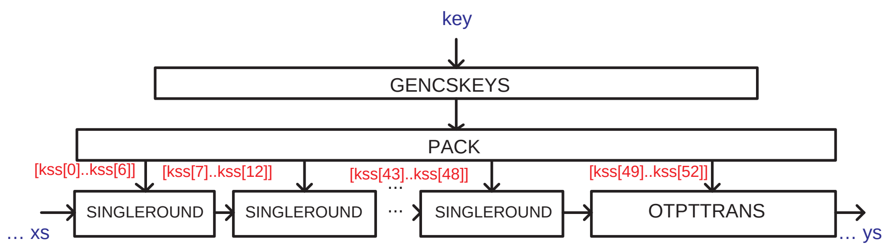

Figure 17: *IDEA* encryption block diagram, a fully-pipelined first design.

The refinement of a single round is a process *SINGLEROUND*. The data refinement realises the inputs and output of this function as vectors of items.

$$singleRound :: \lfloor Int16 \rfloor_4 \rightarrow \lfloor Int16 \rfloor_6 \rightarrow \lfloor Int16 \rfloor_4$$
$$singleRound \sqsubseteq SINGLEROUND$$
$$SINGLEROUND = FIRSTSUBROUND \gg_6 SECONDSUBROUND \gg_6 THIRDSUBROUND$$

where:

$$firstSubRound \sqsubseteq FIRSTSUBROUND$$
$$secondSubRound \sqsubseteq SECONDSUBROUND$$
$$thirdSubRound \sqsubseteq THIRDSUBROUND$$

SINGLEROUND is depicted in Figure 18.

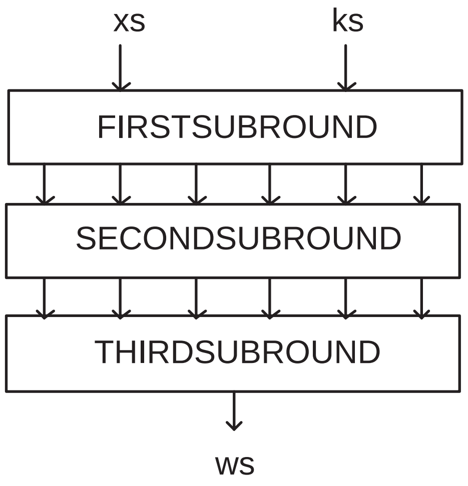

Figure 18: The process *SINGLEROUND* as the piping of the three subrounds.

The refinement of the function *firstSubRound* realises the inputs and output as vectors of items. The refinement of the functions *addMod* and *mulMod* (modular addition and multiplication) are the processes *AddMod* and *MulMod* respectively.

$$firstSubRound :: \lfloor Int16 \rfloor_4 \rightarrow \lfloor Int16 \rfloor_6 \rightarrow \lfloor Int16 \rfloor_6$$
$$FIRSTSUBROUND = |||_{j=0}^{j=1} (ADDINV[in/in_j, out/out_j] \; ||| \; MULINV[in/in_j, out/out_j] \; ||| \; (Forward[in/in_j, out/out_j]))$$

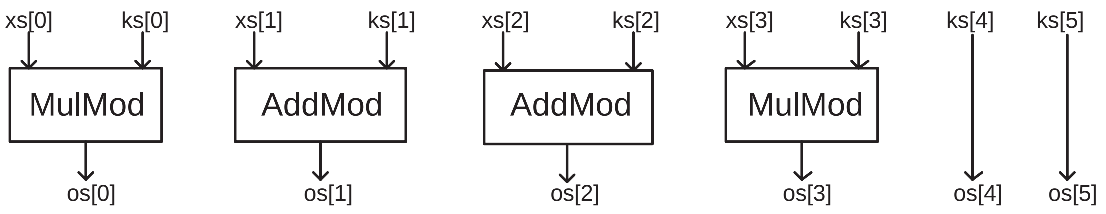

Figure 19: The process *FIRSTSUBROUND*.

Visualization of the process *FIRSTSUBROUND* is shown in Figure 19. The refinement of the function *secondSubRound* is *SECONDSUBROUND*. The *CSP* implementation consider the input and output as vectors of items:

$$secondSubRound :: \lfloor Int16 \rfloor_6 \rightarrow \lfloor Int16 \rfloor_6$$

$$\begin{aligned} SECONDSUBROUND &= (VZIPWITH_2(EXOR))\|MA \\ & EXOR = (in_1?l1 \rightarrow SKIP \mid\mid\mid in_2?l2 \rightarrow SKIP);\ out!(l1 \oplus l2) \\ & fullexor \sqsubseteq EXOR \end{aligned}$$

where, $\oplus$ is the bit-wise execlusive-OR. This process is visualized in Figure 20.

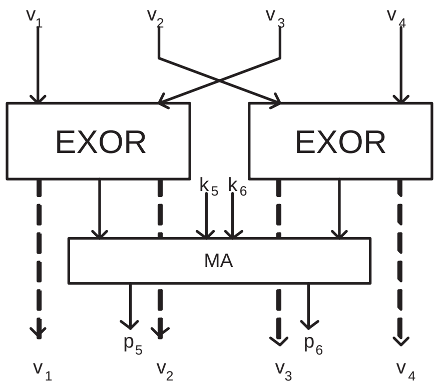

Figure 20: The processes *SECONDSUBROUND*.

The refined process *MA* satisfies the function *mA*. Accordingly, the *CSP* implementation is as follows:

$$mA :: \lfloor Int16 \rfloor_4 \rightarrow (Int16, Int16)$$

$$\begin{aligned} MA = {} & (\mid\mid\mid_{i=1}^{i=4} in_1?list.Elements[i] \rightarrow SKIP); \\ & MULMOD \parallel IBROADCAST_2[d/out] \parallel \\ & (ADDMOD \gg MULMOD) \parallel IBROADCAST_2[d/out] \parallel ADDMOD \end{aligned}$$

The process $IBROADCAST_n$ broadcasts a single input to an *n* independent channels. The Process *MA* is visualized in Figure 21.

Data refinement considers the input and output of the function *thirdSubRound* as vectors of vectors:

$$thirdSubRound :: \lfloor Int16 \rfloor_6 \rightarrow \lfloor Int16 \rfloor_4$$

$$\begin{aligned} THIRDSUBROUND = {} & (\mid\mid\mid_{i=0}^{i=5} in.elements[i]?[y1, y2, y3, y4, p1, p2] \rightarrow SKIP); \\ & (PRD([y1, y3, y2, y4] \parallel PRD([p2, p2, p1, p1] \parallel VZIPWITH_4(EXOR)) \end{aligned}$$

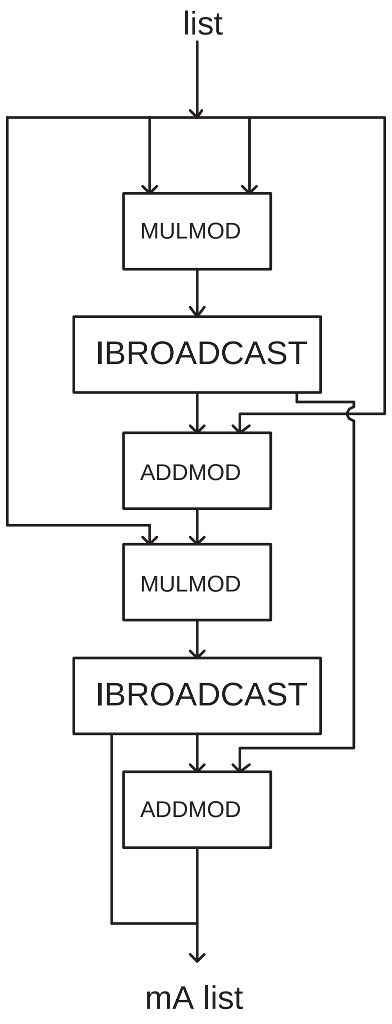

Figure 21: The process *MA*.

Visualization of the process *THIRDSUBROUND* is shown in Figure 22.

The refinement for the *outputTransformation* is done in a similar way to *FIRSTSUB-ROUND*.

7.3.2 IDEA Second Design

As indicated earlier, in this design the generated subkeys communicate with encryption (or decryption) process as a stream of vectors of items as depicted in Figure 23. The refinement for this design is as follows:

$$\begin{aligned}&\mathit{ENCRYPTSEGS} = \\ &\qquad \mathit{GENCSKEYS} \gg_{52} \mathit{PACK} \gg_{9} (\mathit{SVFOLDL}(\mathit{SINGLEROUND}) \| \mathit{OTPTTRANS})\end{aligned}$$

The process *SVFOLDL(SINGLEROUND)* uses an off-the-shelf refinement for the high-order-function *foldl* over a stream of vectors of items.

7.3.3 IDEA Third Design

This design is a compromised solution between the first and the second design. The *CSP* implementation of this design allows the passing of the subkeys in two ways. Some of the

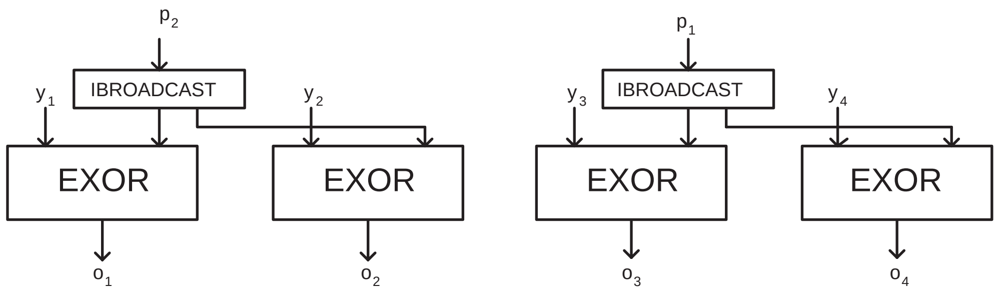

Figure 22: The process *THIRDSUBROUND*.

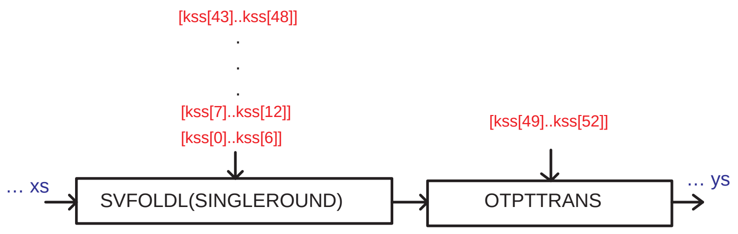

Figure 23: *IDEA* encryption block diagram in its second design with streamed input and output.

subkeys elements are passed as a vector of vectors, while the remaining subkeys are communicated as a stream of vectors. This design is shown in Figure 24.

$$
\begin{aligned}
ENCRYPTSEGS(n) = & GENCSKEYS \gg_{52} PACK \gg_{9} \\
& ((n \triangleright VVFOLDL(SINGLEROUND)) \\
& \| SVFOLDL(SINGLEROUND) \\
& \| OTPTTRANS)
\end{aligned}
$$

The processes *VVFOLDL(SINGLEROUND)* and *SVFOLDL(SINGLEROUND)* are running in parallel synchronising on the output of *VVFOLDL(SINGLEROUND)* to be the input to *SVFOLDL(SINGLEROUND)*. The number of folded processes is produced for each where n is the number of folded processes having subkeys as vector of vectors.

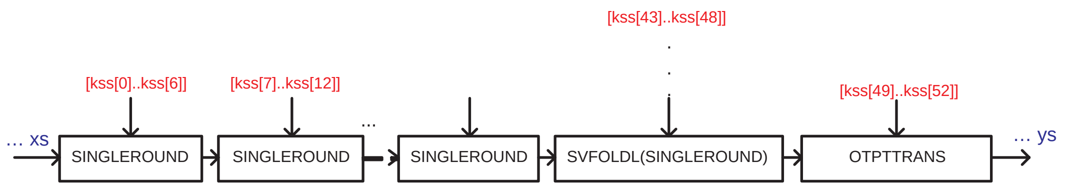

Figure 24: *IDEA* encryption block diagram, partially pipelined third design.

8 Reconfigurable Hardware Realization

In this section we discuss only some pieces of the code implementing the presented designs. The whole implementations were tested and the practical evaluation and analysis is presented in the next section. Coding with *Handel-C*, the structure of the implementation is based on the network of communicating processes given by the refinement. A part of the code implementing the macro *GenerateEncSubKeys* is as follows:

```
par {
    KeyRotation (Key1, Vlst, VoVOut1);
    VMap (VoVOut1, VoVOut2, Size, GenerateSubKeys);
    // The output is concatenated in one vector of vectors VoVOut2
}
```

The used datatypes were declared as:

```
Item (key1, Int128);  // The Key
VectorOfVectorsOfItems (voVOut1, 8, 8, Int16);  // Intermediate
VectorOfVectorsOfItems (voVOut2, 8, 8,Int16);  // Result
```

Another example is the hardware implementation of the second design for decryption subkeys generation. In this code the encryption subkeys are loaded from the memory bank produced as a stream of vectors of 6 elements to a macro *Perform* that performs the required computation based on the *CSP* refinement.

The macro that performs the computation and the main program that uses it are as shown in the following listing.

```
StreamOfVectorsOfItems (sKssIn, 6, Int16);
StreamOfVectorsOfItems (sUssOut, 6, Int16);
.
.
.
par{
    ProduceStreamOfVectorsOfItems (sKssIn, 9, 6, P');
    Map (sKssIn, sUssOut, Perform);
    StoreStreamOfVectorsOfItems (sUssOut, 6, decSubKeyss);
}

macro proc Perform (sKIn, sUOut) {
    par {
        MulInv (sKIn.elements[0], sUOut.elements[0]);
        AddInv (sKIn.elements[1], sUOut.elements[1]);
        AddInv (sKIn.elements[2], sUOut.elements[2]);
        MulInv (sKIn.elements[3], sUOut.elements[3]);
        ForwardItem (sKIn.elements[4], sUOut.elements[4]);
        ForwardItem (sKIn.elements[5], sUOut.elements[5]);
    }
}
```

The implementation for a single round is done by implementing each sub-round as a macro. The macro that corresponds for the process *FIRSTSUBROUND* is:

```
macro proc FirstSubRound (xs, ks, ts) {
    par {
        AddMod (xs.elements[1], ks.elements[1], ts.elements[1]);
        MulMod (xs.elements[0], ks.elements[0], ts.elements[0]);
        AddMod (xs.elements[2], ks.elements[2], ts.elements[2]);
        MulMod (xs.elements[3], ks.elements[3], ts.elements[3]);
        ForwardItem (ks.elements[4], ts.elements[4]);
        ForwardItem (ks.elements[5], ts.elements[5]);
    }
}
```

Thus for a single round:

```
macro proc Round (xsIn, ksIn, wsOut) {
    .
    .
    .
    par {
        FirstSubRound (xsIn, ksIn, one);
        SecondSubRound (one, two);
        ThirdSubRound (two, wsOut);
    }
}
```

Turning our attention to another example, we choose the implementation of the encryption third design. Whereby, a combination of parallel and sequential fold are employed. Comments on the functionality are included near each statement.

```
void main (void) {
    .
    .
    .
    par {
        // Get plaintext.
        ProduceStreamOfVectorsOfItemsFromBank0 (xsSOV, 4);

        // Produce subkeys for VVFoldL
        ProduceVectorOfVectorsOfItems (vVSubKeys, pRnds, 6, subKeyss);
        VVFoldL (vVSubKeys, 6, xsSoVector, 4, pRnds, Round, xsSOV);

        // Produce the remaining subkeys for SVFoldL
        ProduceStreamOfVectorsOfItems (sVSubKeys, sRnds, 6, subKeyss1);
        SVFoldL (sVSubKeys, 6, sWs, Round, xsSoVector, 4, sRnds);

        // Produce subkeys for the output transformation
        ProduceVectorOfItems (ks9, 4, subKeys9);
        OutputTransformation (sWs, ks9, xsSoVector1);

        // Store the ciphertext
        StoreStreamOfVectorsOfItemsInBank1 (xsSoVector1, 4);
    }
}
```

9 Performance Analysis and Evaluation

Generally, the suggested algorithms inherit all the advantages from the development method applied. Key issues are granted, like the production of reusable, scalable, and correct solutions by construction as opposed to trial and testing. Correctness, which is an important aspect in security algorithms, is ensured by construction through the functional specification step. Recall that according to this specification, the implementation under *HUGs98 Haskell* compiler is tested at the unit, component and integration levels.

Table 1 shows the results for the encryption and decryption subkeys generation. Note that the test key used for the generation is the key whose 16-bit segments are: $\{1 ,2 ,3 ,4 ,5 ,6 ,7 ,8\}$. We also recall that the execution time of doing only the handshaking between the host and the *RC-1000* system with no computations costs approximately $132\,\mu$*Sec*. Some of decryption first design's results are marked as not available as the design was too large for the compiled device. The encryption keys are expanded with a throughput of 4.089 Gbps occupying an area of 5846 Slices, i.e. 12% of the area of the available *FPGA*. The speed dramatically goes down with the decryption subkeys expansion using the second stream-based design (6.68 Mbps and an area of 9032 Slices).

Table 1: Results for encryption and decryption subkeys generation.

Metrics \ Designs	Encryption Subkeys Generation	Decryption Subkeys Generation First Design	Decryption Subkeys Generation Second Design
Number of Gates	64906 NANDs	4094334 NANDs	162923 NANDs
Number of Occupied Slices	5846 Slices (12%)	92091 Slices (321% Overmapped)	9032 Slices (47%)
Total equivalent gate count	80784 Gates	1012481 Gates	132128 Gates
Number of Cycles	14 Cycles	88 Cycles	588 Cycles
Maximum Frequency of Design	68.81MHz	NA	4.72 MHz
Throughput	4.089 Gbps	NA	6.68 Mbps
Measured Execution Time	167 Micro sec.	NA	299 Micro sec.
Measured Throughput	23.77 Mbps	NA	4.98 Mbps

Table 2 presents the results for the different designs of encryption (decryption). The findings reflect the change of performance with respect to the change of design. The first design, as intended, is the fastest with a max throughput of 21.33 Gbps (average throughput of 21.5 Mbps) noted from testing random input test vectors with a key = $\{1\ ,2\ ,3\ ,4\ ,5\ ,6\ ,7\ ,8\}$. The second design, which correspond to a sequential execution of the rounds has an expected slowest throughput (maximum throughput of 5.82 Gbps and average throughput of 19.53 Mbps), but the minimum circuit area 5650 Slices (29% of the area of the used *FPGA*). The third design trades the throughput for the used area, thus it has a compromised performance as compared to the first and second designs. Many tests are run using random test vectors and keys to measure the average throughput shown in Table 2. Table 3 shows different ratios relative to a suggested design. For instance, This table shows the Gates Saving Ratio with respect to the second design. This ratio is an indicator for how many times more (or less) a design would use gates taking the second design result as a reference value.

Table 2: Results for encryption (or decryption) for different test vectors.

Metrics \ Designs	1st Fully-Pipelined Design	2nd Stream-Based Design	3rd Partially-Pipelined (2 Parallel and 6 Sequential)
Number of Gates	394526 NANDs	88651 NANDs	176583 NANDs
Number of Occupied Slices	19198 Slices (99%)	5650 Slices (29%)	10147 Slices (52%)
Total equivalent gate count	363682 Gates	93659 Gates	172719 Gates
Number of Cycles / Key ={1,2,3,4,5,6,7,8}	88 Cycles	415 Cycles	382 Cycles
Maximum Frequency of Design	34.975 MHz	44.72 MHz	36.42 MHz
Throughput	25.4 Mbps	6.89 Mbps	6.1 Mbps
Best Measured Execution Time	0.036 Micro Sec.	0.04475 Micro Sec.	0.0425 Micro Sec.
Average Measured Execution Time	2.98 Micro sec.	3.276 Micro sec.	3.086 Micro sec.
Best Measured Throughput	1.777 Gbps	1.430 Gbps	1.505 Gbps
Average Measured Throughput	21.5 Mbps	19.53 Mbps	20.73 Mbps

Different implementations of modular arithmetic operations dramatically affect the performance of the *IDEA*. Three implementations for the modular multiplication are being investigated. The first implementation uses a fast and expensive version of the modulo operator *mod*. A second implementation corresponds to an iterative version of the operation *mod*. The third implementation is for the efficient implementation shown in the specification section,

Table 3: Comparisons among suggested designs.

Metrics \ Designs	First Design	Second Design	Third Design (2 parallel and 6 sequential)
Gates Saving Ration wrt Second Design	3.88 (288% more gates)	N_1 = T E No. Gates (No Comm.) N_2 = T E No Gates (No Comm.) of 2nd Design N_1/N_2 ((N_1 - N_2) / N_2 %)	1.84 (84% more gates)
Number of Cycles Ratio wrt First Design	C_1 = No. Cycles C_2 = No cycles of 1st Design C_1/C_2 (C_1 - C_2) / C_2 %)	4.7159 (371.59% more cycles)	4.3409 (340.9% more cycles)
Best Time Ratio wrt First Design	E_1 = Exec. Time E_2 = Exec. Time of 1st Design E_1/E_2 (E_1 - E_2) / E_2 %)	1.24 (24% more time)	1.18 (18% more time)
Average Time Ratio wrt First Design		1.099 (9.9% more time)	1.035 (3.5% more time)
Best Measured Speedup Ratio wrt First Design	S_1 = Speed of 1st Design S_2 = Speed S_1/S_2 (S_1 - S_2) / S_2 %)	1.24 (24% faster)	1.18 (18% faster)
Average Measured Speedup Ratio wrt First Design		1.1 (10% faster)	1.037 (3.7% more time)
Occupied Area (Slices) Ratio wrt Second Design	3.39 (239% larger area)	A_1 = Area (No Comm.) A_2 = Area (No Comm.) of 2nd Design A_1/A_2 (A_1 - A_2) / A_2 %)	1.77 (77% larger area)

which eliminates the use of the operation *mod*. Table 4 shows comparisons among the suggested implementations of the modular multiplication as used in the second design. This table shows that the efficient implementation of the modular multiplication has affected the performance of the *IDEA* positively. This is shown in the reduced cycle count taken by this implementation as compared to the two other implementations. It also reduced the used area to 5650 Slices after being 6263 and 10739 Slices in the other implementations.

Table 4: Results for encryption second design for different versions of 'mod', for different test vectors.

Second Design Modular Multiplication Implemented using: (Key Used = {1,2,3,4,5,6,7,8})	'mod' Operator	Iterative 'mod'	Efficient Implementation Eliminating 'mod'
Number of Gates	172164 NAND Gates	95226 NAND Gates	93659 NAND Gates
Number of Cycles	988 Cycles	106060 Cycles	415 Cycles
Best Measured Execution Time	0.04475 Micro Sec.	0.15075 Micro Sec.	0.011 Micro sec.
Best Measured Speed	1.430 Gbps	424.54 Mbps	5.82 Gbps
Number of Occupied Slices	10739 (55%)	6263 (32%)	5650 (29%)
Total equivalent gate count	168889	103057	93659

To present some results from the literature for hardware implementations of the *IDEA* algorithm, A summary of findings is shown in Table 5. In [12, 13, 11, 18], the authors present different ad hoc hardware implementations of the *IDEA* algorithm. The *IDEA* block cipher has been implemented at throughput ranging from 8.5 Gbps [13] to 177 Mbps [18] on *FPGAs*. Note that while a 528 Mbps throughput was achieved [14], with a fully pipelined architecture, the implementation required four *Xilinx XC4020 FPGAs*.

Table 5: Comparison among different hardware implementations of the IDEA.

System \ Metrics	Speed (Mbps)	Clock Frequency (MHz)	Area
PCI Pamette - 4 Xilinx XC4020 FPGAs (Zimmermann et al)	528	33	3200 CLBs
UNICORN Architecture; Xilinx FPGA (Runje et al)	2.8	NA	NA
XCV1000-6 Xilinx FPGA (2 Cores) (Cheung et al)	5250	82	11602 Slices
XCV1000-6 Xilinx FPGA (Beuchat et al)	8000	NA	4845 Slices
XCV2000e-6 Xilinx FPGA (Beuchat et al)	8500	NA	18164 Slices
XC2V4000-6 Xilinx FPGA (Beuchat et al)	7900	NA	18537 Slices

10 Conclusion

We investigated in this paper the synthesis of highly parallel reconfigurable hardware implementation for the *IDEA*. Important aspects for hardware implementations of cryptographic algorithms like correctness, reliability along with efficiency are stressed through the application of the proposed development model. The development for the *IDEA* started by formally specifying the algorithm in a functional setting. At that point, provably correct refinement rules are applied transforming the specification to different proposed designs. Thereby, implementations with different levels of parallelism are studied. The refined designs include the blocks from *IDEA* responsible for encryption and decryption in addition to their subkeys generators. The reconfigurable circuits' realization using *Handel-C* is done based on the refined *CSP* networks. The first design requires 88 computing cycles yielding an average throughput of 25.4 Mbps. The maximum throughput achieved with random test vectors was 21.33 Gbps. The second design occupied the minimum area among the different designs with 5650 slices. Currently, our research is concentrating on widening the area of application of the development model, besides, automating the development process.

References

[1] A. E. Abdallah. Derivation of Parallel Algorithms from Functional Specifications to CSP Processes. *Mathematics of Program Construction, LNCS 947, (Springer Verlag, 1995)*, pages 67–96, 1995.

[2] J. Hawkins and A. E. Abdallah. Functional process modelling. *Proceedings of the 7th IEEE International Conference on Electronics, Circuits and Systems*, December 2000.

[3] I. Damaj, J. Hawkins, and A. Abdallah. Mapping high level algorithms onto massively parallel recofigurable hardware. *ACS/IEEE International Conference on Computer Systems and Applications*, pages 14–22, July 2003.

[4] Celoxica. Handel-c documentation. *http://www.celoxica.com/*, 2003.

[5] National Bureau of Standards, U.S. Department of Commerce. *Data encryption standard*, January 1977.

[6] B. Schneier. *Applied Cryptography*. John Wiley and Sons, New York, 1996.

[7] X. Lai and J. Massey. A proposal for a new block encryption standard. In *Proceedings of the EUROCRYPT 90 Conference*, pages 389–904, 1990.

[8] A. Tanenbaum. *Computer Networks*. Prentice Hall, Upper Saddle River, NJ, third edition, 1997.

[9] J. Daemen, R. Govaerts, and J. Vandewalle. Weak keys for IDEA. *Springer-Verlag*, pages 224–231, 1994.

[10] William Stallings. *Network Security Essentials*. Prentice Hall, third edition, November 2002.

[11] D. Runje and M. Kovac. Universal strong encryption FPGA core implementation. In *Proceedings of Design, Automation and Test in Europe*, pages 923–924, 1998.

[12] O. Y. H. Cheung, K. H. Tsoi, P. H. W. Leong, and M. P. Leong. Tradeoffs in parallel and serial implementations of the international data encryption algorithm IDEA. *Lecture Notes in Computer Science*, 2162:333, 2001.

[13] Jean-Luc Beuchat and Jean-Michel Muller. Modulo *m* multiplication-addition: Algorithms and FPGA implementation. *Electronics Letters*, 40(11):654–655, May 2004.

[14] O. Mencer, M. Morf, and M.J. Flynn. Hardware software tri-design of encryption for mobile communication units. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 5, pages 3045–3048, 1998.

[15] Allen Michalski, Kris Gaj, and Tarek El-Ghazawi. An Implementation Comparison of an IDEA Encryption Cryptosystem on Two General-Purpose Reconfigurable Computers. In *Field-Programmable Logic and Applications: 13th International Conference, FPL*, Lecture Notes in Computer Science, pages 204 – 219, Lisbon - Portugal, 2003. Springer.

[16] Jean-Luc Beuchat. Modular multiplication for FPGA implementation of the IDEA block cipher. In Ed Deprettere, Shuvra Bhattacharyya, Joseph Cavallaro, Alain Darte, and Lothar Thiele, editors, *Proceedings of the 14th IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 412–422. IEEE Computer Society, 2003.

[17] Alfred J. Menezes, Paul van Oorschot, and Scott A. Vanston. *Handbook of Applied Cryptography*. CRC Press, fifth edition, August 2001.

[18] R. Zimmermann, A. Curiger, H. Bonnenberg, H. Kaeslin, N. Felber, and W. Fichtner. A 177 MBps VLSI implementation of the International Data Encryption Algorithm. *IEEE Journal of Solid State Circuits*, 29(3):303–307, March 1994.