Falcon Takes Off - A Hardware Implementation of the Falcon Signature Scheme

Michael Schmid[1], Dorian Amiet[1], Jan Wendler[1], Paul Zbinden[1], and Tao Wei[2]

[1] Eastern Switzerland University of Applied Sciences, Oberseestrasse 10, Rapperswil, Switzerland
{michael.schmid2, dorian.amiet, jan.wendler, paul.zbinden}@ost.ch
[2] University of Rhode Island, Kingston, Rhode Island, United States
tao_wei@uri.edu

Abstract. FALCON is one out of three post-quantum signature schemes which have been selected for standardization by NIST in July 2022. To the best of our knowledge, FALCON is the only selected algorithm that does not yet have a publicly reported hardware description that performs signing or key generation. The reason might be that the FALCON signature and key generation algorithms do not fit well in hardware due to the use of floating-point numbers and recursive functions. This publication describes the first hardware implementation for FALCON signing and key generation. To overcome the complexity of the FALCON algorithms, High-Level Synthesis (HLS) was preferred over a hardware description language like Verilog or VHDL. Our HLS code is based on the C reference implementation available at NIST. We describe the required modifications in order to be compliant with HLS, such as rewriting recursive functions into iterative versions. The hardware core at security level 5 requires 45,223 LUTs, 41,370 FFs, 182 DSPs, and 37 BRAMs to calculate one signature in 8.7 ms on a Zynq UltraScale+ FPGA. Security level 5 key generation takes 320.3 ms and requires 100,649 LUTs, 91,029 FFs, 1,215 DSPs, and 69 BRAMs.

Keywords: High-Level-Synthesis, FPGA, FALCON, Post-quantum cryptography

1 Introduction

Digital signatures are a widely used cryptographic tool that enables users to prove that a digital message, document, or software code originates from a specific sender. This ensures, among other things, data integrity during transmission. Traditional signature algorithms suffer from progress in the area of quantum computers. Quantum computers use quantum-mechanical phenomena to solve various mathematical problems that are infeasible for traditional computers. In theory, those quantum computers could break many of the standard public key cryptosystems we are using today [27]. This would compromise the confidentiality and integrity of nearly every form of digital communication.

Post-quantum cryptography (PQC) refers to cryptographic algorithms that are designed to be resistant to attacks from both quantum and classical computers. Small quantum computers work in a research environment, but to successfully attack cryptosystems in use, much larger quantum computers will need to be built. It is almost impossible to predict when such quantum computers will be available, but it is considered inevitable that they will come one day [6], and the internet infrastructure must be prepared for that day [22].

The National Institute of Standards and Technology of the United States of America (NIST) began standardizing PQC in 2017 with 69 algorithm candidates. After three selection rounds, four algorithms were chosen in July 2022 to be standardized. Three of them are digital signatures, and one of them is FALCON [13]: **Fa**st Fourier **l**attice-based **co**mpact signatures over **N**TRU. Compared to the other selected signature schemes, FALCON has the smallest bandwidth (public key size plus signature size) and a fast signature verification algorithm.

FPGAs are highly parallelized devices that can perform multiple operations simultaneously, making them ideal for implementing post-quantum cryptographic algorithms as they require numerous operations [21]. FPGAs are not only used to accelerate the algorithm execution; FPGA implementations have also been used to evaluate PQC algorithms in terms of side-channel attack vulnerability [8,1,26]. This is a critical part of the evaluation of PQC algorithms which will be used for real-world applications.

FPGA implementations for the other post-quantum signature schemes selected for standardization exist. There are publications describing CRYSTALS-DILITHIUM [3] hardware implementations [25,4,32,28] and SPHINCS$^+$ [2] implementations [1,7,30].

Contribution: In the case of FALCON, only the signature verification part has been implemented to date [5,29] fully on hardware. In addition, there is a GPU implementation [20] that describes a parallelized FALCON implementation and a partly hardware-accelerated design [19] where the Gaussian sampling part is executed in the FPGA part of a Zynq device. However, a full hardware implementation of the key and signature generation still lacks. This work aims to address this gap by presenting, to the best of our knowledge, the first full hardware implementation of the FALCON signature and key generation algorithms on an FPGA. The focus lies primarily on how to implement the recursive structures of the FALCON. For transparency, we make the source code publicly available[3].

Paper Organization: In Section 2, we summarize the FALCON algorithms. In Section 3, we explain how to rewrite the recursive tree structures in a way that the high-level-synthesis (HLS) is able to generate synthesizable code in a hardware description language. Implementation results and performance compares are presented in Section 4. Finally, the publication is completed with a discussion and a conclusion.

[3] `https://gitlab.ost.ch/imes/public/papers/FalconHLS`

2 Background

This section starts with a general description of FALCON with a focus on the functions with a recursive structure. The second part contains a summary of existing hardware implementations that are limited to signature verification.

2.1 Falcon

FALCON was developed by combining several works, including the lattice-based signature scheme NTRUSign [16] (the GGH cryptosystems where the first to propose a lattice-based signature scheme [15]), which had a flaw in its signing procedure [23,11]. In 2008, Gentry, Peikert, and Vaikuntanathan proposed a method that fixes the flaw and provides a generic framework for building secure hash-and-sign lattice-based signature schemes [14]. Stehlé and Steinfeld combined the GPV framework with NTRU lattices to create a provably secure NTRUSign [31], while Ducas et al. proposed a practical implementation of the IBE part of the GPV framework over NTRU lattices [10]. Ducas and Prest also proposed a new algorithm to address the slow signing time issue, which FALCON uses to propose a practical lattice-based hash-and-sign scheme [12].

FALCON relies on NTRU lattices established by Hoffstein, Pipher, and Silverman [17]. This allows reducing the keys to polynomials of degree n ($n = 2^k$). Computations are done modulo a monic polynomial ϕ of degree n, which equals $\phi = x^n + 1$. Polynomials are treated as vectors and matrices throughout the algorithm. With a small prime $q \in \mathbb{N}$, let $\mathbb{Z}_q$ be the quotient ring $\mathbb{Z}/q\mathbb{Z}$. NTRU Lattices are constructed with the polynomials $f, g, F, G \in \mathbb{Z}[x]/(\phi)$ and the NTRU equation

$$fG - gF = q \pmod{\phi} \tag{1}$$

When f is invertible, then

$$h = gf^{-1} \pmod{\phi \bmod q}. \tag{2}$$

The key pair generation selects random f and g polynomials with a distribution that yields short vectors. Afterward, the NTRU equation is solved to find the matching F and G. The generated polynomials are then stored in a so-called FALCON tree for which the LDL* decomposition $\mathbf{G} = \mathbf{LDL}^\star$ of the matrix $\mathbf{G} = \mathbf{BB}^\star$ must be computed with

$$\mathbf{B} = \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}. \tag{3}$$

$\mathbf{L}$ is stored in the tree root and diagonal elements D_{ii} of $\mathbf{D}$ are split into matrices $\mathbf{G}_i$. Then a subtree for each $\mathbf{G}_i$ is created, and the process starts recursively down to the bottom of the tree. The function $\mathsf{ffLDL}^\star$ shown in Algorithm 1 describes the process to build the FALCON tree with its recursive structure.

Algorithm 1 $\mathsf{ffLDL}^*(\mathbf{G})$ [13]

```
Require: A full-rank Gram matrix $\mathbf{G} \in \mathsf{FFT}(\mathbb{Q}[x]/(x^n+1))^{2\times 2}$
Ensure: A binary tree $\mathsf{T}$
 1: $(\mathbf{L}, \mathbf{D}) \leftarrow \mathsf{LDL}^*(\mathbf{G})$        ▷ $\mathbf{L} = \left[\begin{array}{c|c} 1 & 0 \\ \hline L_{10} & 1 \end{array}\right], \mathbf{D} = \left[\begin{array}{c|c} D_{00} & 0 \\ \hline 0 & D_{11} \end{array}\right]$
 2: $\mathsf{T}.\mathsf{value} \leftarrow L_{10}$
 3: if $(n = 2)$ then
 4:     $\mathsf{T}.\mathsf{leftchild} \leftarrow D_{00}$
 5:     $\mathsf{T}.\mathsf{rightchild} \leftarrow D_{11}$
 6:     return $\mathsf{T}$
 7: else
 8:     $d_{00}, d_{01} \leftarrow \mathsf{splitfft}(D_{00})$
 9:     $d_{10}, d_{11} \leftarrow \mathsf{splitfft}(D_{11})$
10:     $\mathbf{G}_0 \leftarrow \left[\begin{array}{c|c} d_{00} & d_{01} \\ \hline d_{01}^* & d_{00} \end{array}\right], \mathbf{G}_1 \leftarrow \left[\begin{array}{c|c} d_{10} & d_{11} \\ \hline d_{11}^* & d_{10} \end{array}\right]$
11:     $\mathsf{T}.\mathsf{leftchild} \leftarrow \mathsf{ffLDL}^*(\mathbf{G}_0)$        ▷ Recursive calls
12:     $\mathsf{T}.\mathsf{rightchild} \leftarrow \mathsf{ffLDL}^*(\mathbf{G}_1)$
13: end if
14: return $\mathsf{T}$
```

The signature generation hashes the message and a randomly generated string (nonce) into a polynomial $c \pmod{\phi}$. The signer uses the secret lattice basis (f, g, F, G) to create a pair of short polynomials (s_1, s_2) where

$$s_1 = c - s_2 h \ (\mathrm{mod}\, \phi \,\mathrm{mod}\, q). \tag{4}$$

The signature itself is s_2.

The signature verification recomputes s_1 by himself given the hashed message c, public key h and signature s_2. The signature is valid if (s_1, s_2) is an appropriately short vector.

As trapdoor function, Falcon uses fast Fourier sampling proposed by Ducas *et al.* [9]. Fast Fourier sampling makes use of the Falcon tree and discrete Gaussians over $\mathbb{Z}$. The function ffSampling shown in Algorithm 2 describes the fast Fourier sampling with its recursive structure.

2.2 Falcon Top Functions

The official Falcon API that follows NIST's guidelines in the PQC contest offers three top functions:

`key_gen` Generate a public and private key pair.
`sign_dyn` Generate the signature given the private key and a message.
`verify` Check the validity of the signature when both signature, public key, and message are known

Algorithm 2 $\mathsf{ffSampling}_n(\mathbf{t}, \mathsf{T})$ [13]

```
Require: $\mathbf{t} = (t_0, t_1) \in \mathsf{FFT}(\mathbb{Q}[x]/(x^n + 1))^2$ a Falcon tree $\mathsf{T}$
Ensure: $\mathbf{z} = (z_0, z_1) \in \mathsf{FFT}(\mathbb{Z}[x]/(x^n + 1))^2$
 1: if $n = 1$ then
 2:     $\sigma' \leftarrow \mathsf{T}.value$
 3:     $z_0 \leftarrow \mathsf{SamplerZ}(t_0, \sigma')$
 4:     $z_1 \leftarrow \mathsf{SamplerZ}(t_1, \sigma')$
 5:     return $\mathbf{z} = (z_0, z_1)$
 6: end if
 7: $(l, \mathsf{T}_0, \mathsf{T}_1) \leftarrow (\mathsf{T.value},\ \mathsf{T.leftchild},\ \mathsf{T.rightchild})$
 8: $\mathbf{t}_1 \leftarrow \mathsf{splitfft}(t_1)$
 9: $\mathbf{z}_1 \leftarrow \mathsf{ffSampling}_{n/s}(\mathbf{t}_1, \mathsf{T}_1)$        ▷ First recursive call to $\mathsf{ffSampling}_{n/2}$
10: $z_1 \leftarrow \mathsf{mergefft}(\mathbf{z}_1)$
11: $t_1' \leftarrow t_0 + (t_1 - z_1) \odot l$
12: $\mathbf{t}_0 \leftarrow \mathsf{splitfft}(t_0')$
13: $\mathbf{z}_0 \leftarrow \mathsf{ffSampling}_{n/s}(\mathbf{t}_0, \mathsf{T}_0)$        ▷ Second recursive call to $\mathsf{ffSampling}_{n/2}$
14: $z_0 \leftarrow \mathsf{mergefft}(\mathbf{z}_0)$
15: return $\mathbf{z} = (z_0, z_1)$
```

The `sign_dyn` function expands the private key into the FALCON tree form and calculates the signature. The private key expanding calculation takes approximately half the processing time of `sign_dyn`. If the same private key is used to sign multiple messages, it makes sense to split the function into two parts:

`expand_pk` Calculate the expanded private key given the private key.
`sign_tree` Generate the signature given the expanded private key and the message.

A call to `expand_pk` and `sign_tree` results in the same signature as a single call to `sign_dyn` would. In the case where the size of the key store is less critical, `key_gen` and `expand_pk` can be called once, and many signatures can be generated by multiple calls to `sign_tree`. Therefore, this paper describes implementations to `key_gen`, `expand_pk`, `sign_tree`, and `verify` (without `sign_dyn`) and still claims to be a full FALCON implementation.

2.3 Existing FPGA Implementations of Falcon

Beckwith et al. [5] published a VHDL implementation, and Soni et al. [29] proposed an HLS implementation of FALCONs signature verification. The statements why only the verification algorithm is implemented are the hard-to-implement recursive structure of the FALCON tree and the use of floating-point numbers.

Although it is possible to perform floating-point arithmetic on FPGAs, floating-point processing usually requires more hardware resources and clock cycles compared to integer arithmetic. The main obstacle is, therefore, the recursive functions. We present a solution on how to rewrite the recursive function in the next Section.

3 FPGA Implementation

Our FPGA implementation is based on the Falcon reference C-code submitted to the NIST PQC standardization process. We used Vitis-HLS and Vivado tool from AMD-Xilinx to synthesize and implement the Falcon algorithms on the FPGA.

This Section focuses on how to rewrite the recursive tree structures in a way that the high-level-synthesis (HLS) can generate synthesizable code in a hardware description language.

3.1 Tree Traversal Algorithm

The signature generation algorithm and the computation of the Falcon tree make use of two recursive structures. Both structures need a sufficiently large array, which is allocated before the recursive functions are called. This array is then used throughout the whole function. Listing 1.1 shows the typical structure of such a top function.

```
void foo_top(double *tree, unsigend n){
  double tmp[2*n];
  double *g0, *g1;
  g0 = tmp;
  g1 = g1 + n;
  // call of a recursive function
  foo_rec(tree, g0, g1, n);
}
```

Listing 1.1. Call of a recursive function with the typical memory layout used in the reference implementation

3.2 Tree Traversal without returning Computations

Listing 1.2 shows the tree traversal structure as used in Algorithm 1.

```
void foo_rec1(double *tree, double *g0, double *g1, int n){
  if (n == 1){ // bottom of the tree
    tree[0] = g0[0];
  }
  else{
    foo(g1, g1 + n/2, g0, n);
    // first recursive call
    foo_rec1(tree + n, g1, g1 + n/2, n - n/2,);
    // second recursive call
    foo_rec1(tree + n, g0, g0 + n/2, n - n/2);
  }
}
```

Listing 1.2. First recursive structure

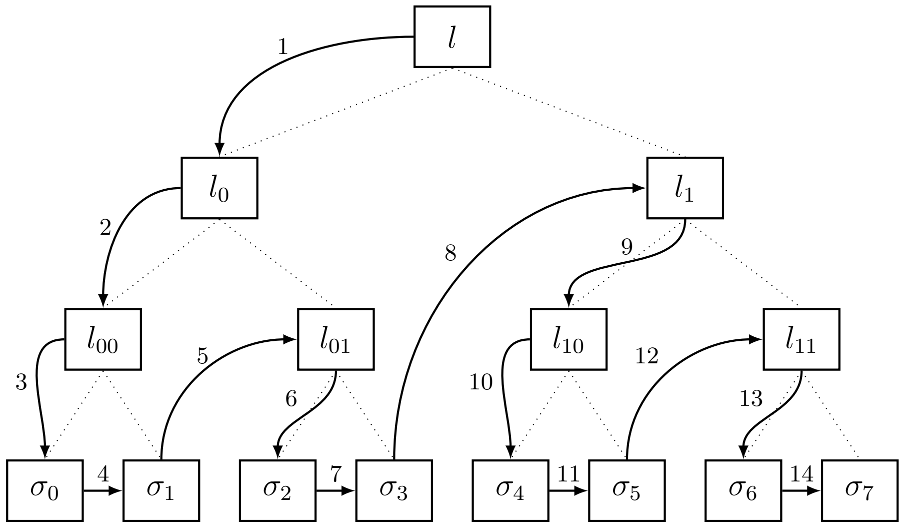

Fig. 1. First recursive structure of the tree traversal algorithm, represented in a binary tree of depth three

Figure 1 shows the tree propagation. Implementing an iterative version of this structure is straightforward. The recursive structure is already resolved with a loop and a look-up table that stores the tree node sequence.

In detail, the number of times the recursive function is called (`n_it`) as well as the value of `n` that has been used for each call (`n_tree`) is evaluated first. To obtain the correct pointer for `g0` and `g1`, the memory offset to the original `tmp` (base address) array needs to be stored. For every recursive function call, the pointer offset from input `g0` and `g1` to `tmp` is then saved in arrays that store these offsets. The same approach is applied to the `tree` input pointer. These precomputed index tables compute the recursive tree structure in a loop as shown in Listing 1.3.

```
void foo_it1(double *tree, double *base_adr){
  for(int i = 0; i < n_it; ++i){
    int n = n_tree[i];
    if (n == 1){ // bottom of the tree
      *(tree+tree_offset[i]) = *(base_adr + g0_offset[i]);
    }
    else{
      foo(base_adr+g1_offset[i], base_adr+g1_offset[i]+n/2,
          base_adr+g0_offset[i], n);
    }
  }
}
```

Listing 1.3. Iterative version of the first recursive structure

This approach has the advantage that it is efficient to implement on FPGAs as these pre-computed arrays can be stored in a few block memories (BRAMs).

3.3 Tree Traversal with returning Computations

The second recursive structure computes something when returning from a recursive call. The ffSampling shown in Algorithm 2 employs this structure. A corresponding code example is presented in Listing 1.4.

```
void foo_rec2(double *tree, double *g0, double *g1, int n){
  if (n == 1){
    tree[0] = g0[0];
  }
  else{
    foo(g1, g1 + n/2, g0, n);
    // first recursive call
    foo_rec2(tree + n, g1, g1 + n/2, n/2,);
    foo(g1, g1 + n/2, g0, n);
    // second recursive call
    foo_rec2(tree + n, g0, g0 + n/2, n/2);
    foo(g1, g1 + hn, g0, n);
  }
}
```

Listing 1.4. Second recursive structure, where something is computed after returning form the recursive call

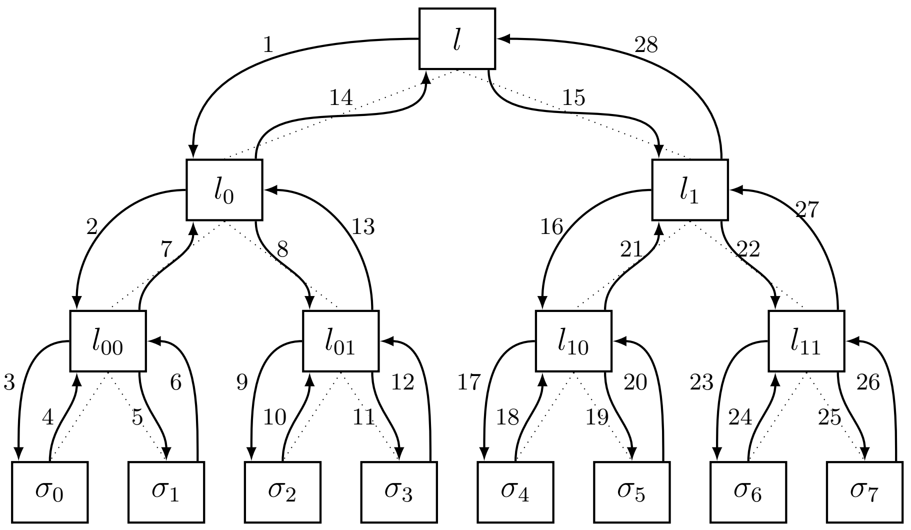

Fig. 2. Second recursive structure of the tree traversal algorithm including back-propagation path, represented in a binary tree of depth three

The difference to the first recursive structure is the back-propagation path. The returning sequence from the recursive calls must be kept. This is illustrated in Figure 2.

```c
void foo_it2(double *tree, double *base_adr){
  int r1_cnt = 0; // How many times returned recursion 1
  int r2_cnt = 0; // How many times returned recursion 2
  for(int i = 0; i<n_it; ++i){
    int n = n_tree[i];
  if (n == 1){ // bottom of the tree
      *(tree+tree_offset[i]) = *(base_adr + g0_offset[i]);
    }
    else{
      foo(base_adr+g1_offset[i], base_adr+g1_offset[i]+n/2,
        base_adr+g0_offset[i], n);
    }
    for(;;){ // there might be multiple returns in a row
      if(i == r2_g[r2_cnt]){// return second recursive call
        double* g0 = base_adr + g0_offset[r2_l[r2_cnt]];
        double* g1 = base_adr + g1_offset[r2_l[r2_cnt]];
        n = n_tree[r2_l[r2_cnt]];
        foo(g1, g1+n/2, g0, n);
        r2_cnt++;
      }
      else{
        break;
      }
    }
    if(i == r1_g[r1_cnt]){ // return first recursive call
      double* g0 = base_adr + g0_offset[r1_l[r1_cnt]];
      double* g1 = base_adr + g1_offset[r1_l[r1_cnt]];
      n = n_tree[r1_l[r1_cnt]];
      foo(g1,g1+n/2, g0, n);
      r1_cnt++;
    }
  }
}
```

Listing 1.5. Iterative version of the second recursive structure

Upon analysis of Figure 2, it can be observed that all branches situated on the left-hand side pertain to the first recursive invocation (e.g., call paths 1, 2, 3, 9, 16, etc. correspond to Listing 1.4 line 9). Analogously, all branches on the right-hand side relate to the second recursive invocation (e.g., call paths 5, 8, 11, 15, etc.) corresponding to Listing 1.4 line 12). Call paths 4, 7, 14, and so forth correspond to the return from the first recursive invocation (Listing 1.4 line 10), whereas call paths 6, 12, 13, and so forth correspond to the return from the second recursive invocation (Listing 1.4 line 13). Notably, multiple consecutive returns may be from the second recursive invocation (12, 13, 14, or 26, 27, 28).

Now, the same approach as in the first structure is followed. Firstly, the offset of the internal data to the base pointer `g0_offset`, `g1_offset`, `tree_offset`, and `n_tree` must be stored.

Additionally, it is necessary to keep track of how many times the function has been called when returning from the first recursive invocation (`r1_g`) and which recursive call belongs to this returning (`r1_l`). This can be observed in Figure 2 with corresponding paths e.g. 1 & 14, 2 & 7, 5 & 6. More precisely, path 14 occurs after 14 recursive calls (`r1_g` = 14). We return from the first recursive invocation, which was called at the 1st recursive call (`r1_l` = 1).

The same applies to the second recursive invocation with corresponding paths, e.g., 5 & 6, 8 & 13, 15 & 28 and the returning information stored in `r2_g` and `r2_l`. Listing 1.5 shows the iterative version of the second recursive structure when all the pre-computed index look-up tables are available.

The recursive structure in `key_gen`, `expand_pk`, and `sign_tree` has been solved with the strategy described above. The second obstacle, the need to handle floating-point numbers, is solved by the tool choice: Floating point arithmetic can be used directly in HLS using common data types such as float or double. HLS utilizes embedded digital signal processors (DSPs) in the FPGA to perform computations with floating-point numbers.

With HLS and the rewritten functions, synthesizable code can already be generated. However, more steps are required to get a better-performing core regarding latency and hardware utilization.

3.4 High-Level Synthesis Optimization

To improve the generation of the hardware description language (HDL) code, pragmas in the C code are used to guide HLS. Here is a selection of these pragmas that we used to guide the HLS in the latency-hardware utilization trade-off:

Array partitioning organizes C arrays in different BRAMs or entirely in FFs instead of a single more significant memory to improve the total memory bandwidth.

Loop unrolling in hardware means that loop iterations are executed in parallel.

Function inlining dissolves functions into the calling function and no longer appears as a separate hierarchy level.

Pipelining allows functions or loops the concurrent execution of operations.

Dataflow enables task-level pipelining, allowing functions and loops to overlap during operation.

During design test and optimization, most of these options increased hardware utilization significantly while the impact on latency reduction was marginal. In the end, only a few of these options are left in the implementation results described in the next section.

Table 1. The reported `sign_tree` and `verify` numbers includes message hashing of a short message (50 bytes)

Function	**Degree**	**BRAM**	**DSP**	**FF**	**LUT**	**Clock Cycles**	**Latency** *ms*	**Clock** *MHz*
`sign_tree`	512	32	182	44,249	46,971	787,441	4.2	187.5
`key_gen`	512	56	1,209	91,615	98,752	†	113.7 ± 22.2	100.0
`expand_pk`	512	23	101	26,083	22,469	544,153	2.5	214.3
`verify`	512	13	15	8,078	11,544	132,482	0.6	214.3
`sign_tree`	1024	37	182	41,370	45,223	1,638,253	8.7	187.5
`key_gen`	1024	69	1,215	91,029	100,649	†	320.3 ± 69.1	100.0
`expand_pk`	1024	29	139	30703	27666	1,191,337	5.6	214.3
`verify`	1024	14	15	8,619	13,302	269,608	1.3	214.3

†Latency of `key_gen` depends on a random seed. Therefore, only the measured latency is given, including standard deviation, but no exact clock cycle numbers.

4 Results

All FALCON functions but `sign_dyn` have been implemented on a Zynq UltraScale+ (ZCU104) FPGA from AMD-Xilinx.

Hardware utilization and latencies of the FALCON functions are shown in Table 1. The number of required clock cycles has been measured with HLS co-simulation, and the maximum clock frequency has been determined with Vivado. All results in this paper are represented in latency. The throughput of a single instance is the reciprocal of the given latency. As the cores could be instantiated several times, the total throughput can be multiplied by the number of instances.

The given clock speeds are chosen so that Vivado's timing analysis is closed (no negative slack remains). The runtime of the key generation process is influenced by the seed value used in the random number generator. As a result, it is not possible to accurately predict the number of clock cycles required using HLS co-simulation. To obtain an estimate of the runtime, the key generation algorithm was executed on our FPGA board with 1000 different seed values, and the average runtime was measured.

4.1 Classification

The most suitable implementation to compare our result are FPGA-based implementations of FALCON. We found two such implementations in the open literature, an HLS implementation [29] and a VHDL implementation [5]. Both only implement the signature verification algorithm. As listed in Table 2, our HLS implementation requires 5 times fewer LUTs and roughly half the latency compared to the HLS implementation from [29]. The latency halving is primarily

Table 2. Hardware Utilization and latency of our core compared to other hard- and software implementations of Falcon and hardware implementations of other PQC signing algorithms.

Algorithm	**Hardware**		**Latency [ms]**		
	Device	**(DSP/BRAM/kFF/kLUT)**	**keygen**	**sign**	**verify**
		Security level 1 - 2			
Falcon-512, our	UltraScale+	(1,209/56/91.6/98.7)	113.7†	-	-
Falcon-512, our	UltraScale+	(101/23/26.1/22.5)	2.5‡	-	-
Falcon-512, our	UltraScale+	(182/32/44.2/47.0)	-	4.2	-
Falcon-512, our	UltraScale+	(15/13/8.0/11.5)	-	-	0.618
Falcon-512 [5]	UltraScale+	(2/4/7.3/14.3)	-	-	0.008
Falcon-512 [29]	Artix-7	(18/26/17.7/57.6)	-	-	0.996
Dilithium-II [5]	UltraScale+	(16/29/28.4/53.9)	0.019	0.117	0.026
SPHINCS^{+}-128f [1]	Artix-7	(1/11.5/72.5/48.0)	-	1.01	0.16
Falcon-512 [24]	Intel i7-6567U	FPU	7.4† + 0.1‡	0.21	0.026
Falcon-512 [18]	Cortex M7	FPU	359† + 6.5‡	13.1	2.6
Falcon-512 [24]	Cortex M4	EMU	1,020† + 96‡	126	3
		Security level 4-5			
Falcon-1024, our	UltraScale+	(1,215/69/91.0/100.6)	320.3†	-	-
Falcon-1024, our	UltraScale+	(139/29/30.7/27.7)	5.6‡	-	-
Falcon-1024, our	UltraScale+	(182/37/41.4/45.2)	-	8.7	-
Falcon-1024, our	UltraScale+	(15/14/8.6/13.3)	-	-	1.258
Falcon-1024 [5]	UltraScale+	(2/4/6.87/13.7)	-	-	0.015
Falcon-1024 [29]	Artix-7	(18/28/18.2/58.6)	-	-	2.1
Dilithium-V [5]	UltraScale+	(16/29/28.4/53.9)	0.055	0.215	0.057
SPHINCS^{+}-256f [1]	Artix-7	(1/22.5/74.5/51.0)	-	2.52	0.21
Falcon-1024 [24]	Intel i7-6567U	FPU	21.6† + 0.2‡	0.26	0.045
Falcon-1024 [18]	Cortex M7	FPU	897† + 14‡	26.9	5.3
Falcon-1024 [24]	Cortex M4	EMU	3,059† + 213‡	268	6.14

†Latency of `key_gen`

‡Latency of `expand_pk`

due to the newer FPGA generation as we used an UltraScale+ FPGA, and the results in [29] are based on an Artix-7. The drawback of the HLS approach compared to a plain HDL implementation (or an HLS implementation started from scratch with an efficient hardware architecture in mind) becomes clear when our implementation is compared to the HDL implementation from [5]. Our implementation is two orders of magnitude slower at similar hardware utilization.

The signature and key generation algorithms cannot be compared to other hardware implementations as we did not find any. However, we can compare the latency to software implementations. Our HLS implementation is three times faster than the Cortex-M7 implementation reported in [18], using a dedicated floating-point unit. Compared to the Cortex-M4 version reported in [24] that does not have a dedicated floating-point unit, our HLS implementation is roughly 30 times faster (except for signature verification where floating-point numbers do not matter). Compared to the software implementation running on the Intel i7-6567U [24], our performance is roughly 30 times slower.

Compared to SPINCS+, another signature algorithm selected for standardization by NIST, our HLS implementation is in the same order of magnitude in terms of hardware utilization and signing time (between 1 and 10 ms latency and roughly 50k LUTs) as the HDL implementation reported in [1]. As SPINCS+ requires primarily hash computations, it fits very well into FPGAs. For CRYSTALS-DILITHIUM, the third signature algorithm selected for standardization by NIST, our HLS implementation signature generation core requires a similar amount of hardware as the HDL implementation from [5], but the Dilithium signature generation is roughly 40 times faster. In addition, the HDL implementation from [5] implements all functions (key generation, signing, and verification) within the same hardware core while we need separate cores.

5 Discussion

The C reference implementation of FALCON is not optimal for hardware implementation. Nonetheless, the Vitis HLS tool handles floating-point numbers and our rewritten iterative loops that emulate FALCONs recursive structure correctly. This results in a functional HDL code that implements the FALCON algorithm bit-by-bit identical to the C reference code. Due to the HLS approach based on the C reference code, the resulting hardware architecture could be better optimized.

For example, our HLS implementation instantiates the ChaCha20 cipher block three times in the signature generation block. The reason for this remains unclear. Technically, it makes little sense as parallel execution is excluded because all ChaCha20 calls require output data from the previous call. In the end, these redundant instances increase the hardware utilization of this part by almost a factor tree at literally zero gain in the latency.

A pure HDL or restructured HLS implementation (HLS code written from scratch, which describes an optimized hardware architecture in more detail) could more precisely address the strengths of FPGAs or hardware in general. It would generate an implementation with improved latency and/or smaller hardware utilization.

5.1 Real Wold Applications

While we were able to implement the complex key generation algorithm on an FPGA, its enormous hardware utilization and longer latency (compared to the Intel i7 software-based implementation) make it somewhat impractical. As the key generation is usually less used than signing and verification, our key generation HLS implementation is probably not attractive for FPGA integration for most applications.

For signing, an FPGA integration might be attractive for some applications as the latency is significantly lower than the Cortex-M7 implementation at an acceptable cost of FPGA resources. An interesting question is how efficient an HDL or newly structured HLS implementation of the signature generation would be. The signature verification HDL implementation from [5] is 100 times faster than our HLS implementation at similar hardware utilization. The open question is if a similar speed-up would be possible for an HDL or newly structured HLS implementation of the signing algorithm. The most expensive parts of our implementation are the calculation in the tree traversal algorithms. These calculations require roughly 80 % of the clock cycles and include the use of floating-point numbers. While the use of floating-point numbers might not be the main issue, speeding up the tree traversal requires complex memory management that allows a kernel to read and write from different addresses in parallel. Further research is required to pipeline the tree traversal structure. The second computationally expensive part that requires roughly 15 % of the clock cycles is the iFFT calculation. This should not be a problem to speed up significantly for a plain HDL or newly structured HLS implementation.

6 Conclusion

This publication presents, to our knowledge, the first FPGA implementation for Falcon signing and key generation, representing an essential step in furthering the NIST standardization process. We present a solution on how to implement the recursive Falcon structures into hardware. The performance of our FPGA implementation is compared to other FPGA implementations of PQC signature schemes selected for standardization. The hardware utilization and latency are in the same region as a SPHINCS$^+$ HDL implementation for signing. However, compared to CRYSTALS-DILITHUM HDL implementations, our HLS implementation needs more resources and has higher latency. Nonetheless, our Falcon core is significantly faster than the CPU versions for embedded devices, even when the CPU uses a dedicated floating-point unit.

References

1. Amiet, D., Leuenberger, L., Curiger, A., Zbinden, P.: Fpga-based sphincs$^+$ implementations: Mind the glitch. In: 23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020. pp. 229–237. IEEE (2020). https://doi.org/10.1109/DSD51259.2020.00046
2. Aumasson, J.P., Bernstein, D.J., Beullens, W., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.L., Hülsing, A., Kampanakis, P., Kölbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P., Westerbaan, B.: SPHINCS+ Stateless hash-based signatures. https://sphincs.org/ (April 2023)
3. Bai, S., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-DILITHIUM. https://pq-crystals.org/dilithium/index.shtml (April 2023)
4. Beckwith, L., Nguyen, D.T., Gaj, K.: High-performance hardware implementation of crystals-dilithium. In: International Conference on Field-Programmable Technology, (IC)FPT 2021, Auckland, New Zealand, December 6-10, 2021. pp. 1–10. IEEE (2021). https://doi.org/10.1109/ICFPT52863.2021.9609917
5. Beckwith, L., Nguyen, D.T., Gaj, K.: High-performance hardware implementation of lattice-based digital signatures. IACR Cryptol. ePrint Arch. p. 217 (2022), https://eprint.iacr.org/2022/217
6. Bernstein, D.J., Buchmann, J., Dahmen, E.: Post-Quantum Cryptography. Springer, Dordrecht (2008). https://doi.org/10.1007/978-3-540-88702-7
7. Berthet, Q., Upegui, A., Gantel, L., Duc, A., Traverso, G.: An area-efficient sphincs$^+$ post-quantum signature coprocessor. In: IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021, Portland, OR, USA, June 17-21, 2021. pp. 180–187. IEEE (2021). https://doi.org/10.1109/IPDPSW52791.2021.00034
8. De Mulder, E., Buysschaert, P., Ors, S., Delmotte, P., Preneel, B., Vandenbosch, G., Verbauwhede, I.: Electromagnetic analysis attack on an fpga implementation of an elliptic curve cryptosystem. In: EUROCON 2005 - The International Conference on "Computer as a Tool". vol. 2, pp. 1879–1882 (2005). https://doi.org/10.1109/EURCON.2005.1630348
9. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-dilithium: A lattice-based digital signature scheme. IACR Trans. Cryptogr. Hardw. Embed. Syst. pp. 238–268 (2018). https://doi.org/10.13154/tches.v2018.i1.238-268
10. Ducas, L., Lyubashevsky, V., Prest, T.: Efficient identity-based encryption over NTRU lattices. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8874, pp. 22–41. Springer (2014). https://doi.org/10.1007/978-3-662-45608-8_2
11. Ducas, L., Nguyen, P.Q.: Learning a zonotope and more: Cryptanalysis of ntrusign countermeasures. In: Wang, X., Sako, K. (eds.) Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7658, pp. 433–450. Springer (2012). https://doi.org/10.1007/978-3-642-34961-4_27

12. Ducas, L., Prest, T.: Fast fourier orthogonalization. In: Abramov, S.A., Zima, E.V., Gao, X. (eds.) Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2016, Waterloo, ON, Canada, July 19-22, 2016. pp. 191–198. ACM (2016). https://doi.org/10.1145/2930889.2930923
13. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon - Fast-Fourier Lattice-based Compact Signatures over NTRU . https://falcon-sign.info/ (April 2023)
14. Gentry, C., Peikert, C., Vaikuntanathan, V.: Trapdoors for hard lattices and new cryptographic constructions. In: Dwork, C. (ed.) Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008. pp. 197–206. ACM (2008). https://doi.org/10.1145/1374376.1374407
15. Goldreich, O., Goldwasser, S., Halevi, S.: Public-key cryptosystems from lattice reduction problems. In: Jr., B.S.K. (ed.) Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings. Lecture Notes in Computer Science, vol. 1294, pp. 112–131. Springer (1997). https://doi.org/10.1007/BFb0052231
16. Hoffstein, J., Howgrave-Graham, N., Pipher, J., Silverman, J.H., Whyte, W.: NTRUSIGN: digital signatures using the NTRU lattice. In: Joye, M. (ed.) Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2612, pp. 122–140. Springer (2003). https://doi.org/10.1007/3-540-36563-X_9
17. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: Buhler, J. (ed.) Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1423, pp. 267–288. Springer (1998). https://doi.org/10.1007/BFb0054868
18. Howe, J., Westerbaan, B.: Benchmarking and analysing the NIST PQC finalist lattice-based signature schemes on the ARM cortex M7. IACR Cryptol. ePrint Arch. p. 405 (2022), https://eprint.iacr.org/2022/405
19. Karabulut, E., Aysu, A.: A hardware-software co-design for the discrete gaussian sampling of falcon digital signature. Cryptology ePrint Archive, Paper 2023/908 (2023), https://eprint.iacr.org/2023/908, https://eprint.iacr.org/2023/908
20. Lee, W.K., Zhao, R.K., Steinfeld, R., Sakzad, A., Hwang, S.O.: High throughput lattice-based signatures on gpus: Comparing falcon and mitaka. Cryptology ePrint Archive, Paper 2023/399 (2023), https://eprint.iacr.org/2023/399, https://eprint.iacr.org/2023/399
21. Li, H., Tang, Y., Que, Z., Zhang, J.: Fpga accelerated post-quantum cryptography. IEEE Transactions on Nanotechnology **21**, 685–691 (2022). https://doi.org/10.1109/TNANO.2022.3217802
22. Mosca, M.: Cybersecurity in an era with quantum computers: Will we be ready? IEEE Security & Privacy **16**(5), 38–41 (2018). https://doi.org/10.1109/MSP.2018.3761723
23. Nguyen, P.Q., Regev, O.: Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures. In: Vaudenay, S. (ed.) Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4004, pp. 271–288. Springer (2006). https://doi.org/10.1007/11761679_17

24. Pornin, T.: New efficient, constant-time implementations of falcon. IACR Cryptol. ePrint Arch. p. 893 (2019), `https://eprint.iacr.org/2019/893`
25. Ricci, S., Malina, L., Jedlicka, P., Smékal, D., Hajny, J., Cíbik, P., Dzurenda, P., Dobias, P.: Implementing crystals-dilithium signature scheme on fpgas. In: Reinhardt, D., Müller, T. (eds.) ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021. pp. 1:1–1:11. ACM (2021). `https://doi.org/10.1145/3465481.3465756`
26. Rodriguez, R.C., Bruguier, F., Valea, E., Benoit, P.: Correlation electromagnetic analysis on an FPGA implementation of crystals-kyber. IACR Cryptol. ePrint Arch. p. 1361 (2022), `https://eprint.iacr.org/2022/1361`
27. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994. pp. 124–134. IEEE Computer Society (1994). `https://doi.org/10.1109/SFCS.1994.365700`
28. Soni, D., Basu, K., Nabeel, M., Aaraj, N., Manzano, M., Karri, R.: CRYSTALS-Dilithium, pp. 13–30. Springer International Publishing, Cham (2021). `https://doi.org/10.1007/978-3-030-57682-0_2`
29. Soni, D., Basu, K., Nabeel, M., Aaraj, N., Manzano, M., Karri, R.: FALCON, pp. 31–41. Springer International Publishing, Cham (2021). `https://doi.org/10.1007/978-3-030-57682-0_3`
30. Soni, D., Basu, K., Nabeel, M., Aaraj, N., Manzano, M., Karri, R.: SPHINCS+, pp. 141–162. Springer International Publishing, Cham (2021). `https://doi.org/10.1007/978-3-030-57682-0_9`
31. Stehlé, D., Steinfeld, R.: Making ntruencrypt and ntrusign as secure as standard worst-case problems over ideal lattices. IACR Cryptol. ePrint Arch. p. 4 (2013), `http://eprint.iacr.org/2013/004`
32. Zhao, C., Zhang, N., Wang, H., Yang, B., Zhu, W., Li, Z., Zhu, M., Yin, S., Wei, S., Liu, L.: A compact and high-performance hardware architecture for crystals-dilithium. IACR Trans. Cryptogr. Hardw. Embed. Syst. pp. 270–295 (2022). `https://doi.org/10.46586/tches.v2022.i1.270-295`