WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI
POLITECHNIKA WROCŁAWSKA

KIERUNEK: INFORMATYKA TECHNICZNA

Zastosowanie procesora RISC-V do szyfrowania i deszyfrowania danych metodą CRYSTALS-Kyber

PROJEKT

Autor: Konrad Pempera

Dr. hab. Jarosław Pempera

Wrocław, czerwiec 2023

Spis treści

1 Cel i zakres projektu

Celem projektu jest zaimplementowanie na procesorze RISC-V szyfrowania i deszyfrowania danych metodą CRYSTALS-Kyber. Cel projektu został zrealizowany w następujących krokach.

1. opracowanie i zaimplementowanie najistotniejszych składowych metody CRYSTALS-Kyber w języku programowania C++, w szczególności algorytmów NTT(number-theoretic transform), INTT(inverse number-theoretic transform),
2. wyszczególnienie najbardziej czasochłonnych obliczeń celem późniejszego zrównoleglenia,
3. zaprojektowanie i zaimplementowanie w języku C++ prostego symulatora procesora Risk V realizującego obliczenia wektorowe
4. implementację algorytmów w asemblerze procesora Risc V i weryfikacja poprawności obliczeń na symulatorze.

Dokumentacja projektu składa się z:

- opis najistotniejszych elementów metody szyfrowania Crystal-Cyber, w szczególności opisu transformat NTT oraz INTT
- projektu i implementacji podstawowwych funkcji wykorzystywanych do implementacji wolnej i szybkiej transformaty NTT i INTT
- opisu sposobu zrównoleglenia najbardziej czasochłonnych obliczeń w transformacie NTT,
- opisu implementacji podstawowych funkcji symulujących obliczenia wektorowe na procesorze Risk V
- implementacje w języku Asembler dla procesora Risc V algorytmu NTT.

W realizacji projektu inspirowano się pracą [11]. Z braku możliwości odtworzenia w pełni środowiska sprzętowego Autorów pracy postanowiono zbadać możliwości implementacji NTT na procesorach Risc V dostępnych obecnie na rynku odtwarzając najistotniejsze elementy algorytmy i/lub architektury zaproponowane w omawianej pracy.

Jako główny cel projektu przyjęto opracowanie algorytmu implementującego NTT (INTT) przy wykorzystaniu rozszerzeń wektorowych istniejących obecnie procesorów Risc V z rozszerzeniami wektorowymi.

Oryginalnym osiągnięciem projektu jest opisanie sposobu implementacji algorytmu NTT (INTT) przy wykorzystaniu wbudowanych operacji wektorowych Risc V oraz wykorzystaniu indeksacji zaproponowanej w pracy [11] w:

- języku C++ na własnoręcznie zaimplementowanym symulatorze Risc V z rozszerzeniami wektorowymi oraz
- w języku asembler Risc V na jednym z dostępnych publicznie symulatorze tego procesora.

W rozdziale 2 przedstawiono krótki opis głównego wyzwania bezpieczeństwa danych oraz opis najistotniejszych osiągnięć zawartych w pracy [11]. W rozdziale 3 opisano naiwną i szybką transformatę NTT i INTT oraz najistotniejszą z punktu widzenia projektu, szybką transformatę w wersji wektorowej. W opisie ostatniej szczegółowo przedstawiono potrzebę i sposób tworzenia wektorów indeksujących. Natomiast w rozdziale 4 opisano najistotniejsze cechy i funkcje przetwarzania wektorowego na procesorze Risc V. Rozdział 5 opisuje przebieg procesu implementacji.

2 Wstęp

Wykorzystywane w praktyce algorytmy kryptograficznych (z kluczem publicznym) w większości przypadków już dzisiaj nie są odporne na ataki wysokowydajnych komputerów (klastrów komputerowych) wyposażonych w procesory wielordzeniowe i/lub programowalne karty graficzne. Pewnym, zwiększeniem bezpieczeństwa takich algorytmów odbywa się poprzez zwiększenie długości klucza (publicznego i prywatnego). Kolejnym zagrożeniem jest pojawienie się komputerów kwantowych, które teoretycznie będą(są) w stanie złamać zabezpieczenie szyfrowanej wiadomości (znaleźć klucz prywatny) w ułamkach sekund. W tym wypadku nawet zwiększenie długości klucza może okazać się niewystarczające, co więcej zwiększająca się z biegiem czasu moc komputerów kwantowych wymagałaby nieustającego wydłużania długości klucza.

Kryptografia postkwantowa (PQC) jest sposobem na zapewnienie bezpieczeństwo zaszyfrowanych danych przed próbami rozszyfrowania przy użyciu komputerów kwantowych. Jedną z najlepszych tego typu metod jest Crystal-Kyber. Zgodnie z naszą wiedzą metoda ta jest odporna na ataki komputerów kwantowych. Crystal-Kyber jest jednym z finalistów trzeciej rundy konkursu na algorytmy postkwantowej kryptografii. Nazwa Kyber pochodzi od fikcyjnych kryształów kyber używanych do zasilania mieczy świetlnych w filmie Gwiezdne Wojny.

Służy do ustanowienia wspólnego klucza między dwiema komunikującymi się stronami bez możliwości odszyfrowania go przez atakującego w systemie transmisji. Ten asymetryczny kryptosystem wykorzystuje wariant problemu uczenia się z błędami jako podstawową funkcję zapadni.

Największym wyzwaniem w implementacji Crystal-Kyber jest implementacja NTT (number-theoretic transform), która wymaga stosunkowo dużej pamięci i złożonego schematu dostępu do niej. Prace nad zrównolegleniem obliczeń realizowanych w NTT na różnego rodzaju procesorach(wspomagających obliczenia wektorowe) i kartach graficznych pojawiają się dopiero od kilku lat. Dla przykładu prace [13] [8] [7] dotyczą implementacji na kartach graficznych, natomiast prace [9] [12] dotyczą implementacji na procesorach w technologii SSE lub AVX. Zgodnie z naszą wiedzą praca [11] jest pierwszą pracą przedstawiającą możliwość implementacji NTT na RISC V.

Autorzy pracy [11] zbadali możliwość implementacji algorytmu NTT w wersji zrównoleglonej na akcelatorze AMD Alveo U250. Akcelerator posiada 12 288 procesorów DSP, 2 688 bloków RAM, 1 728 000 bramek logicznych, 791000 bloków LUTRAM, które mogą być zaprogramowane do składowania danych, 3 456 000 przerzutników, 1344 BUFG (bloki wzmacniające sygnał zegarowy). Taka specyfikacja znacząco przewyższa najlepsze procesory oraz karty graficzne. [1]

RISC V jest stosunkowo ubogą implementacją procesora. Jego potęga opiera się na rozszerzeniach, czyli na dodatkowych instrukcjach i funkcjach dodawanych w celu rozszerzenia możliwości obliczeniowych oraz zwiększenia wydajności procesora. Risc V w podstawowej wersji umożliwia jedynie zarządzanie rejestrami, przesunięcia bitowe, skoki, przykładowe operacje arytmetyczne (dodawanie, odejmowanie, mnożenie całkowitoliczbowe), przerwania oraz zarządzanie pamięcią. Pomimo małego wieku projektu istnieje już wiele rozszerzeń do tego procesora, są to między innymi operacje na liczbach zmiennoprzecinkowych czy operacje wektorowe. Ze względu na otwartą architekturę każdy może doimplementować do procesora swoje rozszerzenia.

W pracy [11] autorzy również rozszerzają procesor RISC V o 10 swoich rozszerzeń. Zostały one umieszczone wraz z krótkim opisem w Tabeli 1. Podczas opisywania funkcji używana jest fraz coeff_data. Jest to pamięć dostępu natychmiastowego, która jest charakterystycznym elementem karty FPGA. Proces odczytu danych z tej pamięci następuje w czasie $O(1)$.

Duża liczba procesorów DSP oraz duża pojemność pamięci LUTRAM pozawala na operowanie na wektorach znacznie większych niż występujące w najlepszych procesorach Risc-V. Z tego względu wykonanie operacji wektorowych na wektrorach składających się z 256, 512 oraz 1024 elementów (występujących w różnych standardach Crystals-Kyber) nie stanowi większego problemu.

3 NTT

NTT jest transformatą analogiczną do transformaty Fouriera, jest używana do wykonywania szybkich operacji na dużych liczbach całkowitych (na dużej liczbie liczb całkowitych) wykorzystujących operacje modulo dla pewnego modułu, który jest stały dla tych obliczeń. Danymi wejściowymi procedury NTT jest wektor składający się z nieujemnych liczb całkowitych o długości n, moduł q oraz generator r. Na Rysunku 1 przedstawiono pseudokod naiwnej implementacji NTT. Złożoność obliczeniowa algorytmu

Tabela 1: Rozszerzenia Risc V [11]

Typ instrukcji	**Instrukcja**	**Opis**	Latencja
Ładowanie wektora	vlpolye8/16/32	Ładuje wektor z pamięci RAM do coeff_data, o długości 8/16/32 bitów	1+VL
Zapisywanie wektora	vspolye8/16/32	Zapisuje wektor z pamięci coeff_data do RAM, o długości 8/16/32 bitów	1+VL
Konfiguracja przesunięcia	vnttcfg	ustawia przesunięcie odczytu danych z wektora składującego indeksy dla NTT	2
Konfiguracja przesunięcia	vinttcfg	ustawia przesunięcie odczytu danych z wektora składującego indeksy dla INTT	2
Konfiguracja przesunięcia	vcwcfg	ustawia przesunięcie odczytu danych z wektora składującego indeksy dla CWM	2
Wczytanie wielomianu	vreadpoly	wczytuje wektor z coeff_data do rejestru wektorowego	LMUL+1
Zapisanie wielomianu	vwritepoly	zapisuje wektor składowany w rejestrze wektorowym do coeff_data	LMUL+1
Wczytanie (twiddle)	vreadtw	wczytuje czynnik	LMUL+1
Dodawanie	vaddmod	dodaje dwa elementy	LMUL+1
Odejmowanie	vsubmod	odejmuje dwa elementy	LMUL+1
Redukcja modulo	vmod	wykonuje operacje modulo na liczbie	LMUL+1
Dzielenie przez dwa	vdivby2	Dzieli wyjście o indeksach motylkowych przez dwa	LMUL+1

wynosi $O(n^2)$, pod warunkiem, że wartości ($r^k \mod q$) zostaną wcześniej obliczone i zapamiętane w pamięci komputera w postaci wektora liczb całkowitych. Na Rysunku 2 przedstawiono pseudokod szybkiej transformaty NTT, który został opracowany na podstawie pseudokodu z pracy [11]. Jest on adaptacją algorytmu Cooleya-Tukeya zamieszczonego w artykule [10]. Algorytm cechuje się złożonością $O(n\ logn)$.

W przypadku szybkiej transformaty wymagany jest złożony dostęp do danych zawartych w przekształcanym wektorze v. Najistotniejsze obliczenia realizowane w krokach 8 - 11 wymagają odwołania się do elementu wektora w znajdującego się na pozycji $j - tej$ i $j + m - tej$ oraz $k - tej$ potęgi pierwiastka (pamiętanej w wektorze). O ile wartość j zmienia się o 1, o tyle wartość $j + m$ oraz k zależna jest od wartości zmiennych w iteracjach z linii 4 i 5. Z tego powodu zastosowanie klasycznych operacji na wektorach takich jak dodawanie, mnożenie, dzielenie całkowitoliczbowe itp., w których w/w operacje wykonywane są na danych znajdujących się w tych samych miejscach w wektorach będących argumentami, w implementacji algorytmu NTT nie jest bezpośrednio możliwe. Współczesne procesory wspomagające obliczenia wektorowe pozwalają na obejście tego problemu poprzez odczyt elementów w kolejności zgodnej z argumentami, które są umieszczone w dodatkowym wektorze (sterującym). Efektywność tego rozwiązania zależy od maksymalnej długości wektora obsługiwanego przez mechanizm przetwarzania równoległego w danym procesorze. Przed przystąpieniem do omówienia głównej idei procesu obliczeń równoległych w algorytmie NTT zaproponowanym przez autorów pracy [11] zauważmy następujące fakty:

- dla każdego i liczba kroków od 8 do 12 jest identyczna i wynosi $n/2$,
- dla ustalonego i w każdej iteracji po s i po j para (j, $j + m$) jest różna.

Z drugiego faktu wynika, że po odczycie elementów wektora na pozycjach j i $j + m$ i następnie ich modyfikacji wartości te nie będą odczytywane i modyfikowane w kolejnych iteracjach po s i j. Wartości te będą odczytywane i modyfikowane dopiero w kolejnej iteracji po i.

W zaproponowanym rozwiązaniu zrównolegleniu podlegają obliczenia wykonane w pętlach po s oraz j, tj. kod zawarty w liniach od 5 do 14. Podczas obliczeń realizowanych w krokach od 8 do 13 istotne są następujące indeksy elementów w wektorze v oraz k: j, $j + m$ dla wektora v oraz dla wektora R, w którym zapisane są wartości $r^k \mod q$. Niech $\_j[iter]$ ($\_jm[iter]$, $\_k[iter]$) będzie wartością

Wejście: Tablica v o rozmiarze n zawierająca nieujemne wartości całkowite, q (moduł), r (pierwiastek).
Wyjście: Zaktualizowana tablica v po zastosowaniu transformacji NTT.

```
1  Function NaiveNTT(v, r, q):
2      for i ← 0 to n − 1 do
3          sum ← 0;
4          for j ← 0 to n − 1 do
5              k ← (i · j) mod n;
6              sum ← (v[j] · (r^k mod q) + sum) mod q;
7          end
8          output[i] ← sum;
9      end
10     for i ← 0 to n − 1 do
11         v[i] ← output[i];
12     end
```

Rysunek 1: Naiwny algorytm wyznaczający NTT

Wejście: Tablica v o rozmiarze n zawierająca nieujemne wartości całkowite, q (moduł), r (pierwiastek).
Wyjście: Zaktualizowana tablica v po zastosowaniu transformacji NTT.

```
1  Function NTT(v, r, q):
2      l ← log_2(n)
3      m ← 1
4      for i ← 0 by 1 to l − 1 do
5          for s ← 0 by m · 2 to 2^l − 1 do
6              k ← 0
7              for j ← s by 1 to s + m do
8                  a ← v[j]
9                  b ← (v[j + m] · (r^k mod q)) mod q
10                 v[j] ← (a + b) mod q
11                 v[j + m] = (a − b + q) mod q
12                 k ← k + n/(m · 2)
13             end
14         end
15         m ← m · 2
16     end
```

Rysunek 2: Szybki algorytm wyznaczający NTT

Wejście: Tablica v o rozmiarze n zawierająca nieujemne wartości całkowite, q (moduł), r (pierwiastek).
Wyjście: Zaktualizowana tablica v po zastosowaniu transformacji NTT.

```
1  Function NTT(v, r, q):
2      l ← log_2(n)
3      for i ← 0 by 1 to l − 1 do
4          gather(R, R̄, _k + i · m/2)
5          gather(v, ā, _j + i · m/2)
6          gather(v, b̄′, _jm + i · m/2)
7          vmul(b̄′, R̄, b̄)
8          vmodreduction(b̄, b̄, q)
9          vadd(ā, b̄, c̄)
10         vmodreduction(c̄, c̄, q)
11         vsub(ā, b̄, d̄)
12         vmodreduction(d̄, d̄, q)
13         gather(c̄, v, _jm + i · m/2)
14         gather(d̄, v, _j + i · m/2)
15     end
```

Rysunek 3: Szybki algorytm wyznaczający NTT - wersja wektorowa

zmiennej j ($j + m$, k) w *iter* iteracji algorytmu. Na Rysunku 3 przedstawiono wersję równoległą algorytmu NTT. Przyjmując, że każda z instrukcji z linii od 4 do 14 wykonywana jest w jednym takcie obliczeń wektorowych procesora, można zauważyć, że algorytm cechuje się złożonością obliczeniową $O(log_2 n)$. W kolejnych krokach algorytmu wykonywane są następujące operacje:

- 4 - gather - załadowanie do wektora $\bar{R}$, $n/2$ wartości z wektora R, indeksowanych pozycjami z wektora $\_k$ od pozycji $i \cdot n/2$,
- 5 - gather - załadowanie do wektora $\bar{a}$, $n/2$ wartości z wektora v, indeksowanych pozycjami z wektora $\_j$ od pozycji $i \cdot n/2$,
- 6 - gather - załadowanie do wektora $\bar{b'}$, $n/2$ wartości z wektora v, indeksowanych pozycjami z wektora $\_jm$ od pozycji $i \cdot n/2$,
- 7 - vmul - mnożenie wektora $\bar{b'}$ przez $\bar{R}$ i zapisanie wyniku w wektorze $\bar{b}$,
- 8 - vmodreduction - dzielenie wartości elementów wektora $\bar{b}$ mod q i zapisanie wyniku w wektorze $\bar{b}$,
- 9 - vadd - dodanie wektorów $\bar{a}$ oraz $\bar{b}$ i zapisanie w wektorze $\bar{c}$,
- 10 - vmodreduction - dzielenie wartości elementów wektora $\bar{c}$ mod q i zapisanie wyniku w wektorze $\bar{c}$,
- 11 - vsub - odjęcie wektora $\bar{b}$ od $\bar{a}$ i zapisanie wyniku w wektorze $\bar{d}$,
- 12 - vmodreduction - dzielenie wartości elementów wektora $\bar{d}$ mod q i zapisanie wyniku w wektorze $\bar{d}$,
- 13 - gather - załadowanie do wektora v, $n/2$ wartości wektora $\bar{c}$, indeksowanych pozycjami z wektora $\_jm$ od pozycji $i \cdot n/2$,
- 14 - gather - załadowanie do wektora v, $n/2$ wartości wektora $\bar{d}$, indeksowanych pozycjami z wektora $\_j$ od pozycji $i \cdot n/2$.

4 Risc V

Risc-V jest opensource'ową architekturą procesora ISA. Jest to alternatywa dla architektur takich jak x86. Zawiera rejestry ogólnego przeznaczenia oraz rejestry specjalne. Ogólne rejestry służą do przechowywania danych, a specjalne do kontroli działania procesora oraz przechowywania adresów pamięci. Risc-V umożliwia rozszerzenie podstawowego zestawu instrukcji. Jednym z przykładów rozszerzeń jest rozszerzenie Vector Extension v, na którym opiera się praca badawcza [11]. Rozszerzenie to wprowadza wiele instrukcji, które zwiększają wydajność w przypadku operacji na wielu elementach danych jednocześnie. Obsługiwane jest kilka rodzajów danych tj. operacje na liczbach zmiennoprzecinkowych, całkowitoliczbowych. Oferowany jest również szereg operacji logicznych, operacji arytmetycznych, operacji kontrolujących pamięć oraz porównań. Ze względu na dostępność wielu narzędzi oraz kompilatorów opartych na Risk-V, jest on architekturą rozwijaną i bardzo popularną w środowiskach naukowych oraz badawczych.

Obliczenia wektorowe dokonywane są na dodatkowej puli rejestrów, standardowo są to 32 rejestry, każdy z nich w zależności od specyfikacji może mieć rozmiar (długość 128, 256, 512, 1024 - bitów). Wielkość rejestru zależy od sprzętowej implementacji procesora Risc-V. Np. w zestawie SiFive Performance P600-Series zastosowany jest procesor Risc-V, w którym stosowane są rejestry wektorowe o długości 128 bitów [2].

Najważniejszymi pojęciami związanymi z opisem wektorów są:

- SEW (SEW= 8, 16, 32, 1024) - określa liczbę bitów na których zapisana jest dana w wektorze,
- VLEN - rozmiar rejestru wektorowego w bitach,
- VLMAX - VLEN/SEW - liczba elementów w rejestrze wektorowym,
- LMUL (LMUL = 1, 2, 4, 8) - multiplikator długości wektora, który umożliwia połączenie kilku rejestów w wektor, składający się z LMUL rejestrów wektorowych w jeden.

Przykład. W naszym projekcie będziemy używali wektorów, których elementy są 32-bitowe oraz używali procesora RISC - V, z 128 - bitowymi rejestrami wektorowymi. SEW wynosi 32, VLEN wynosi 128, natomiast VLMAX wynosi 4 (128/32). Dla LMUL = 1 liczba elementów w wektorze wynosi 4, natomiast dla LMUL = 8, liczba elementów wynosi $LMUL \cdot VLEN = 32$.

W Tabeli 2 zamieszczono podstawowe rozkazy procesora Risc-V, które są szczególnie przydatne w przypadku projektowania algorytmów opartych o działania na wektorach w architekturze Risc-V. Pogrubioną czcionko napisano instrukcje, które zostały wykorzystane do implementacji algorytmu NTT o rozmiarze 32 bitów. Są to klasyczne operacje wektorowe takie jak vadd.vv - dodawanie wektorów, vsub.vv - odejmowanie wektorów, vmul.vv - mnożeni wektorów, vremu.vx - reszta z dzielenia wektora przez skalar oraz specyficzne dla procesora RISC V instrukcje vsetvli, vlxwu.v, vsxw.v. Instrukcja vsetvli konfiguruje wektory, gdzie parametrem rs1 jest żądana liczba elementów wektora, a rd to przydzielona długość wektora (zależy od parametrów wewnętrznych procesora RISC V (VMUL, VLEN)) oraz imm zawierający informacje o typie kodowania. Instrukcja vlxwu.v jest instrukcją kopiowania elementów z pozycji składowanych w wektorze będącym trzecim argumentem tego rozkazu, wektora będącego drugim argumentem do wektora będącego pierwszym argumentem. Instrukcja vsxw.v jest instrukcją kopiowania elementów z wektora będącego drugim argumentem na pozycje określone w wektorze trzecim w wektorze będącego pierwszym argumentem.

Obliczenia wykonywane na wektorach w transformacie NTT wymagają tzw. dostępu "motylkowego"do danych, tj. wyznaczenie pewnej wartości wymaga dostępu do danych znajdujących się na dwóch różnych pozycjach w wektorze. Jest to jedna z głównych przyczyn w trudności w zrównolegleniu obliczeń wykonywanych w ramach tej transformacji. Rozszerzenie wektorowe RISC V umożliwia operowanie na dużych wektorach, w naszym przypadku dla VMUL = 8, VLMAX = 4, SEW = 32, otrzymujemy wektor o długości 1024 bity, które pozwala nam na operowanie na wektorach składających się z 32 elementów 32 bitowych. Długość wektora znacznie przewyższa długość rejestrów stosowanych chociażby w takiej technologii jak SSE czy AVX. Jednakże najistotniejszymi instrukcjami potrzebnymi do implementacji NTT są instrukcje vlxwu.v oraz vsxw.v, które umożliwiają wyżej omówiony dostęp do danych.

Tabela 2: Wybrane instrukcje wektorowe Risc V

Insrtukcja	Opis
Set vector configuration	
rd final length, rs1 application vector length, imm encode type	
vsetvli rd, rs1, imm	
vsetvli rd, rs1, imm, VLMUL	
Vector indexed loads	
vd destination, rs1 base address, vs2 indices	
vlxb.v vd, (rs1), vs2, vm	8b
vlxh.v vd, (rs1), vs2, vm	16b
vlxw.v vd, (rs1), vs2, vm	32b
vlxbu.v vd, (rs1), vs2, vm	8b unsigned
vlxhu.v vd, (rs1), vs2, vm	16b unsigned
vlxwu.v vd, (rs1), vs2, vm	**32b unsigned**
Vector ordered-indexed store instructions	
vs3 store data, rs1 base address, vs2 indices	
vsxb.v vs3, (rs1), vs2, vm	8b
vsxh.v vs3, (rs1), vs2, vm	16b
vsxw.v vs3, (rs1), vs2, vm	**32b**
Vector unordered-indexed store instructions	
vsuxb.v vs3, (rs1), vs2, vm	8b
vsuxh.v vs3, (rs1), vs2, vm	16b
vsuxw.v vs3, (rs1), vs2, vm	**32b**
vsuxe.v vs3, (rs1), vs2, vm	SEW
Integer adds.	
vadd.vv vd, vs2, vs1, vm	**Vector-vector**
vadd.vx vd, vs2, rs1, vm	vector-scalar
vadd.vi vd, vs2, imm, vm	vector-immediate
Integer subtract	
vsub.vv vd, vs2, vs1, vm	**Vector-vector**
vsub.vx vd, vs2, rs1, vm	vector-scalar
Signed multiply, returning low bits of product	
vmul.vv vd, vs2, vs1, vm	**Vector-vector**
vmul.vx vd, vs2, rs1, vm	vector-scalar
Signed multiply, returning high bits of product	
vmulh.vv vd, vs2, vs1, vm	Vector-vector
vmulh.vx vd, vs2, rs1, vm	vector-scalar
Unsigned divide.	
vdivu.vv vd, vs2, vs1, vm	Vector-vector
vdivu.vx vd, vs2, rs1, vm	vector-scalar
Signed divide	**Signed divide**
vdiv.vv vd, vs2, vs1, vm	Vector-vector
vdiv.vx vd, vs2, rs1, vm	vector-scalar
Unsigned remainder	
vremu.vv vd, vs2, vs1, vm	Vector-vector
vremu.vx vd, vs2, rs1, vm	**vector-scalar**

```
class NTTDesine {
    public:
    int findModulus(int vector[], int n) {}

    int reciprocal(int n, int mod) {}

    int findPrimitiveRoot(int totient, int mod, int n) {}
};
```

Rysunek 4: Definicja klasy NNTDesine

```
    class NTT {
public:
    int n;
    int  *output;
    int q;
    int r;
    int reprocal;

    NTT(int n, int r, int q, int reprocal);
    virtual void transform(int v[], int r) {
        for (int i = 0; i < n; i++) {
            int sum = 0;
            for (int j = 0; j < n; j++) {
                int k = (int)((long)i * j % n);
                long temp = (long)v[j] * pow(r, k, q) + sum;
                sum = (int)(temp % q);
            }
            output[i] = sum;
        }
        for (int i = 0; i < n; i++) {
            v[i] = output[i];
        }
    }

    virtual void inverseTransform(int v[], int r) {
        transform(v, reprocal);
        int scaler = reciprocal(n, q);
        for (int i = 0; i < n; i++) {
            v[i] = (int)((long)v[i] * scaler % q);
        }
    }
};
```

Rysunek 5: Implementacja naiwnej NTT

5 Realizacja projektu

5.1 Implementacja naiwnej i szybkiej transformaty NTT

Pierwszym etapem realizacji projektu było zaimplementowanie naiwnej oraz szybkiej transformaty NTT. Transformata NTT wymaga następujących parametrów:

- moduł q,

- generator r,
- n.

Zautomatyzowano proces doborów parametrów r i q implementując klasę NTT Desine, której definicję podano w lstlistingu 4. Funkcja findModulus wyznacza moduł q, findPrimitiveRoot wyznacza generator r, reciprocal wyznacza odwrotność $r \mod q$. Liczba q jest liczbą pierwszą i musi być większa od maksymalnego elementu znajdującego się w wektorze. r to taka liczba, gdzie $r^{q-1} \mod q = 1$ i $r^{(q-1)/p} = 1$ dla każdego czynnika pierwszego p wynikającego z podziału na liczniki pierwsze q. Złożoność obliczeniowa funkcji transform wynosi $O(n^2)$ i wynika z podwójnego zapętlenia (po i oraz j).

Na Rysunku 5 znajduje się definicja klasy NTT, która zawiera funkcję transform, jest ona naiwną implementacją transformaty NTT. Funkcja inverseTransform, jest odwrotną transformatą NTT. Polami tej klasy są pola q, r, n, które muszą zostać podane podczas tworzenia obiektu tej klasy (w kodzie źródłowym na Rysunku 5 nie przedstawiono konstruktora inicjującego te parametry).

Na Rysunku 6 przedstawiono kod implementujący szybką transformatę NTT. Implementacja korzysta z dwóch tablic alokowanych dynamicznie, które przechowują potęgi liczb, które są wykorzystywane przez szybką transformatę, jedna tablica dotyczy zwykłej transformaty, natomiast druga dotyczy odwrotnej transformaty. W tablicach tych składowane są potęgi r, które są wyliczane w konstruktorze, gdyż nie zależą od wartości elementów wektora, a wyłącznie zależą od q, r oraz *reprocal*. Funkcja transform jest ww. transformatą NTT, a funkcja inverseTransform jest odwrotną transformatą. Dodatkowo w implementacji występuje funkcja transform, która zawiera dwa argumenty. Została ona stworzona w celu nadpisania funkcji klasy nadrzędnej NTT i możliwości wywołania jej jak w przypadku klasy nadrzędnej.

```
    class FastNTT : public NTT {
public:
    int* powTable;
    int* powTableReverse;

    FastNTT(int n, int root, int mod, int reprocal) :NTT(n, root, mod, reprocal) {
        createPowTable(root, mod, powTable);
        createPowTable(reprocal, mod, powTableReverse);
     }

    void transform(int vector[], int root) {
        transform(vector, root, powTable);
    }

    virtual void inverseTransform(int v[], int root) {
        transform(v, reprocal, powTableReverse);
        int scaler = reciprocal(n, mod);
        for (int i = 0; i < n; i++) {
            v[i] = (int)((long)v[i] * scaler % mod);
        }
    }

    void transform(int vector[], int root, int* powTable) {
        int levels = static_cast<int>(std::log2(n));
        bit_reverse(vector, output);
        for (int i = 0; i < n; i++)  vector[i] = output[i];

        int m = 1;
        for (int i = 0; i < levels; i++) {
            for (int s = 0; s < power(2, levels); s += m * 2) {
                int k = 0;
                for (int j = s; j < s + m; j++) {
                    long a = vector[j];
                    long b = (long)vector[j + m] * powTable[k] % mod;
                    vector[j] = (int)((a + b) % mod);
                    vector[j + m] = (int)((a - b + mod) % mod);
                    k += n / (m * 2);
                }
            }
            m *= 2;
        }
    }
};
```

Rysunek 6: Implementacja szybkiej transformaty NTT

5.2 Implementacja szybkiej transformaty NTT przy wykorzystaniu symulatora Risc V z rozszerzeniami wektorowymi

W celu oceny możliwości zastosowania procesora RISC V z rozszerzeniami wektorowymi do implementacji NTT, zaprojektowano i zaimplementowano klasę implementującą najistotniejsze funkcje wektorowe procesora RISC V. Zmienne ra ... zero znajdujące się w liniach od 3 do 7 oznaczają nazwy rejestrów procesora RISC V, natomiast od v0 do v7 są to nazwy wektorów, których struktura składa się z elementów o SEW = 32, VLMAX = 4 i LMUL 8. Rozkaz vwritepoly służy do przepisania wartości z pamięci do wektora, natomiast rozkaz vwritepoly służy do przepisania wartości z wektora do pamięci. Pozostałe rozkazy zostały omówione w Rozdziale 3. Na Rysunku 7 przedstawiono implementacje symulatora RISC V.

Na Rysunku 8 znajduje się funkcja wyznaczająca NTT, która jest częścią symulatora i korzysta z jego symulowanych rozkazów. Główna pętla w funkcji NTT wykonywana jest $levels$ razy ($2^{levels} = n$). Przyjmując, że każda instrukcja wektorowa wykonywana jest przez procesor RISC V w czasie $O(1)$ (nie jest to spełnione w symulacji, gdzie wynosi to $O(n/2)$) wykonanie każdej iteracji będzie zajmowało 16 jednostek czasu, zatem czas wykonania funkcji będzie rzędu $log_2(n)$.

Na Rysunku 9 znajduje się kod klasy w której uruchamiany jest algorytm NTT zaimplementowany w symulatorze RISC V. Znajdują się tam funkcje takie jak indexing, która wypełnia tablice _j, _jm, _k odpowiednimi wartościami, które następnie są używane jako indeksy podczas wykonywania funkcji korzystających z dostępu "motylkowego". Znajduje się tam również funkcja transform, która wywołuje funkcję NTT z symulatora RISC V z uprzednio utworzonymi parametrami.

```
    class RiscV {
public:
    long v0[32], v1[32],v2[32],v3[32],v4[32],v5[32],v6[32], v7[32];
    long ra, sp, gp, tp, t0, t1, t2, s0, s1, a0,
    a1, a2, a3, a4, a5, a6, a7,
    s2, s3, s4, s5, s6, s7, s8, s9, s10, s11,
    t3, t4, t5, t6, zero=0;
public:
   void vlpolye32(long vd[], int memory[], int vm) {
        for (int m = 0; m < vm; m++)  vd[m] = memory[m];
    }
    void vwritepoly(long rs1[], int memory[], int vm) {
        for (int m = 0; m < vm; m++)  memory[m] = rs1[m];
    }
    //Vector indexed loads
    void vlxwu_v(long vd[], long rs1[],  long vs2[], int vm) {
        for (a3 = 0; a3 < vm; a3++) vd[a3] = rs1[vs2[a3]];
    }
    //Vector ordered-indexed store instructions
    void vsxw_v(long vs3[], long rs1[],  long vs2[], int vm) {
        for (a3 = 0; a3 < vm; a3++) vs3[vs2[a3]] = rs1[a3];
    }
    //Integer adds
    void vadd_vv(long vd[], long vs2[], long vs1[],  int vm) {
        for (a3 = 0; a3 < vm; a3++) vd[a3] = vs2[a3] + vs1[a3];
    }
    //Integer subtract
    void vsub_vv(long vd[], long vs2[], long vs1[],  int vm) {
        for (a3 = 0; a3 < vm; a3++) vd[a3] = vs2[a3] - vs1[a3];
    }
    //Signed multiply, returning low bits of product
    void vmul_vv(long vd[], long vs2[], long vs1[],  int vm) {
        for (a3 = 0; a3 < vm; a3++) vd[a3] = vs2[a3] * vs1[a3];
    }
    //Unsigner remainder - vector - scalar
    void vremu_vx(long vd[], long vs2[], int rs1, int vm) {
        for (a3 = 0; a3 < vm; a3++) vd[a3]=(vs2[a3]+rs1) % rs1;
    }
};
```

Rysunek 7: Implementacja Symulatora RISK V

```
void NTT(int vector[], int n , int levels, int _j[], int _jm[],int _k[], int powTabl
{
  vlpolye32(v0, vector, n); //v0 wektor
  for (t3 = levels; t3 >0; t3--)
    {   int offset = (levels - t3) * n / 2;
        vlpolye32(v2, _jm + offset, n / 2);//v2 = _jm
        vlxwu_v(v5, v0, v2, n / 2); //v5 = b1 = vector[jm]
        vlpolye32(v1, _k + offset, n / 2);//G4 = k = _k[iter]
        vlpolye32(v4, powTable, n / 2);//v6 = powertable
        vlxwu_v(v7, v4, v1, n / 2);//v1 = powertable[k]
        vmul_vv(v5, v5, v7, n / 2);
        vremu_vx(v5, v5, mod, n / 2);
        vlpolye32(v7, _j + offset, n / 2); //v1 indeksy j
        vlxwu_v(v3, v0, v7, n / 2);//v4 = a = vector[j]
        vadd_vv(v4, v3, v5, n / 2);//v4=(a+b)%mod
        vremu_vx(v4, v4, mod, n / 2);
        vsxw_v(v0, v4, v7, n / 2);
        vsub_vv(v4, v3, v5, n / 2);//v4=(a-b)%mod
        vremu_vx(v4, v4, mod, n / 2);
        vsxw_v(v0, v4, v2, n / 2);
    }
  vwritepoly(v0, vector, n);
}
```

Rysunek 8: Sekcja wyznaczająca NTT w symulatorze RISK V

5.3 Szybka transformata NTT w asemblerze Risc V

W ramach realizacji zadania projektowego opracowano kod w assemblerze w składni MIPS. Program został napisany oraz przetestowany w symulatorze Ripes w wersji v2.2.5-10-g976190d. Implementacja składa się z trzech sekcji:

1. definicja danych,
2. kod właściwy NTT,
3. funkcje implementujące rozkazy wektorowe.

Kod assemblerowy funkcji NTT został przedstawiony na Rysunku 10 oraz 9. Ze względu na dużą objętość kod definiujący dane oraz funkcje wektorowe zostały przeniesione do dodatku i znajdują się na Rysunkach 12 **??** oraz 15.

W sekcji danych deklarowane są wszystkie dane. Są to:

- V - wektor danych, jego wartości ulegają transformacie NTT,
- PowerTab - tablica zawierająca potęgi r, odpowiednik PowTab z wcześniejszych rozdziałów,
- _j, _jm, _k - tablice zawierające indeksy,
- od v0 do v7 - miejsce w pamięci przeznaczone do pamiętania wektorów
- q - moduł,
- str1 oraz str2 - służą do formatowania łańcucha znaków, który jest używany podczas wyświetlania wyniku,
- offset - przesunięcie adresu odczytu danych tablic zawierających indeksy w celu ich przepisywania do wektorów.

```
class FastNTTWektor : public NTT {
public:
    int* powTable, powTableReverse, *_j, *_jm, *_k, levels;
    FastNTTWektor(int n, int root, int mod, int reprocal) :NTT(n, root, mod, reprocal) {
        levels = static_cast<int>(std::log2(n));
        createPowTable(root, mod, powTable);
        createPowTable(reprocal, mod, powTableReverse);
        indexing();
    }
    void transform(int vector[], int root) { transform(vector, root, powTable); }
    void transform(int vector[], int root, int* powTable) {
        bit_reverse(vector, output);
        for (int i = 0; i < n; i++)  vector[i] = output[i];
        RiscV riscV;  riscV.NTT(vector, n, levels, _j, _jm, _k, powTable, mod);
    }
    void indexing() {
        int iter = 0, m = 1;
        for (int i = 0; i < levels; i++) {
            for (int s = 0; s < power(2, levels); s += m * 2) {
                int k = 0;
                for (int j = s; j < s + m; j++) {
                    _j[iter] = j;  _jm[iter] = j + m;  _k[iter] = k;
                    iter++;
                    k += n / (m * 2);
                }
            }
            m *= 2;
        }
    }
};
```

Rysunek 9: Wywołania symulacji

Przesyłanie argumentów do funkcji realizujących podstawowe operacje wektorowe realizowane jest poprzez rejestry procesora RISC V. Przyjęto, że rejestr a3 przechowuje długość wektora, a4 identyfikuje wektor źródłowy, w przypadkach gdzie funkcja posiada tylko jeden argument źródłowy a5 identyfikuje wektor docelowy, a6 identyfikuje wektor z indeksami, a w przypadku gdy funkcja korzysta z dwóch adresów źródłowych, wektor a5 pełni funkcje adresu źródłowego, a wektor a6 pełni funkcję wektora docelowego. Nazwy etykiet identyfikujących fragmenty kodu realizującego funkcje wektorowe zostały nazwane zgodnie z nazewnictwem rozkazów wektorowych.

Kod implementujący funkcje wektorowe został przedstawiony na Rysunkach **??**, 15, podano tam również wymaganą zawartość rejestrów opisujących argumenty tych funkcji.

```
.globl _start
_start:

        addi s3, zero, 5 ; levels
        vwritepoly v0, V, offset0, n ;vwritepoly(v0, vector, n); //v0 wektor
        ;for (s3 = levels; s3 >0; s3--) {
ll:
        addi a0, zero, 5
        sub a0, a0, s3 ;a0 = levels - s3
        addi s4, zero, 64
        mul a0, a0, s4
        la a1, offset
        sw a0, 0(a1) ;offset = (levels - s3) * n/2

        vwritepoly v2, _jm, offset, nby2 ;vwritepoly(v2, _jm + offset, n / 2);//v2 = _jm

        vlxwu_v v5, v0,  v2 ;vlxwu_v(v5, v0, v2, n / 2); //v5 = b1 = vector[jm]

        vwritepoly v1, _k,  offset, nby2 ; vwritepoly(v1, _k + offset, n / 2);//G4 = k =

        vwritepoly  v4, PowerTab, offset0, nby2 ; vwritepoly(v4, powTable, n / 2);//v6 =

        vlxwu_v v7, v4, v1 ; vlxwu_v(v7, v4, v1, n / 2);//v1 = powertable[k]

        vmul_vv v5, v5,  v7 ; vmul_vv(v5, v5, v7, n / 2);

        vremu_vx  v5, mod ; vremu_vx(v5, v5, mod, n / 2);

        vwritepoly  v7, _j, offset, nby2 ; vwritepoly(v7, _j + offset, n / 2); //v1 indel

        vlxwu_v  v3, v0,  v7

        vadd_vv  v4, v3, v5 ; vadd_vv(v4, v3, v5, n / 2);//v4=(a+b)%mod

        vremu_vx  v4, mod ; vremu_vx(v4, v4, mod, n / 2);

        vsxw_v  v0, v4,  v7 ; vsxw_v(v0, v4, v7, n / 2);

        vsub_vv  v4, v3, v5 ; vsub_vv(v4, v3, v5, n / 2);//v4=(a-b)%mod

        vremu_vx  v4, mod ; vremu_vx(v4, v4, mod, n / 2);

        vsxw_v  v0, v4, v2 ; vsxw_v(v0, v4, v2, n / 2);

        addi s3, s3, -1
        bne s3, zero, ll

        vwritepoly V, v0,  offset0, n ; vwritepoly(v0, vector, n);

        printResult V

        ; Exit program
        li a7, 10
        ecall
```

Rysunek 10: NTT w asemblerze Risc V

5.4 Testowanie w środowiskach symulacyjnych Risk V

Symulator Ripes w wersji v2.2.5-10-g976190d, który został użyty do testowania programu w assemblerze, oferuje wiele przydatnych funkcji dla użytkowników. Praca w trybie krokowym, gdzie każda instrukcja jest wykonywana w ustalonym czasie, daje możliwość dokładnej analizy działania programu i zobaczenia, która instrukcja jest aktualnie wykonywana. To przydatne narzędzie zarówno do nauki jak i debugowania kodu.

Dodatkowo, możliwość edycji, kompilacji, uruchomienia aplikacji oraz zapisu i wczytania kodu źródłowego do pliku sprawia, że Ripes jest elastycznym narzędziem, które ułatwia testowanie, eksperymentowanie i usprawnianie kodu.

Jako projekt o otwartej architekturze, Ripes jest wciąż rozwijany i rozbudowywany o dodatkowe funkcjonalności.

Niestety, ze względu na to, że Ripes jest projektem wciąż młodym oraz brakiem szerokiej implementacji procesorów RISC-V obsługujących operacje wektorowe, symulacja operacji wektorowych nie jest dostępna w tym symulatorze. Wspomniane operacje wektorowe wymagają specjalnego sprzętu i wsparcia procesora, które nie są obecnie uwzględnione w zakresie funkcjonalności Ripes.

Na Rysunku 11 przedstawiono zrzut ekranu wykonany podczas wykonywania programu, który został zaimplementowany podczas wykonywania zadania projektowego. Symulacja odbywała się w trybie krokowym. W lewej stronie okna można zauważyć kod źródłowy. W środkowym oknie umieszczony jest zdeassemblowany kod, a czerwona poświata wskazuje, która instrukcja jest aktualnie wyświetlana. Po prawej stronie w skrajnym panel wypisane są rejestry procesora RISC V, a obok nich wyświetlane są ich wartości. W dolnej części symulatora znajduje się konsola. Na niej za pomocą funkcji print wyświetlany jest wynik transformaty.

Środowisko Ripes nie wspiera kompilacji macr, w związku z tym docelowy program dla procesora RISC V został skompilowany("riscv64-unknown-elf-as projektSIMD2.s -o projektSIMD2.o") oraz zlinkowany("riscv64-unknown-elf-ld projektSIMD2.o -o projektSIMD2 –static –no-dynamic-linker –no-omagic –no-relax") przy pomocy risc-v-gcc. Wynikiem kompilacji jest między innymi plik z rozszerzeniem elf, który następnie został wczytany do środowiska Ripes i uruchomiony w celu sprawdzenia poprawności działania programu.

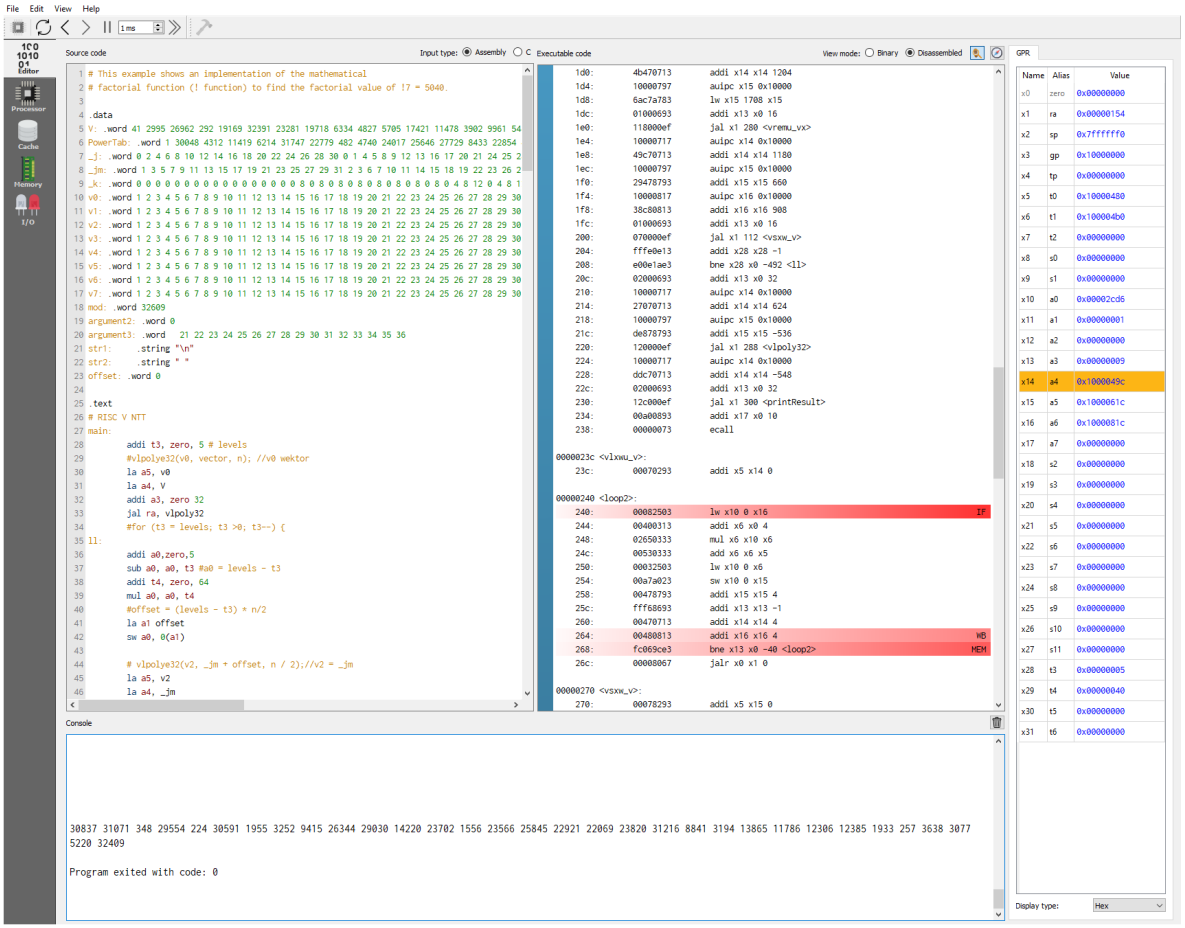
6 Wnioski

Podczas realizacji zadania projektowego w pierwszej kolejności zaimplementowaliśmy algorytm szybkiej transformaty NTT. Implementacja została napisana w języku C++ oraz w assemblerze dla RISC V. Dodatkowo zaimplementowano symulator w języku C++, który symuluje rozkazy wektorowe, które zaoferowali autorzy pracy [11].

Praca nad projektem ukazała jak wielki potencjał ma procesor RISC V. Podczas wykonywania projektów dowiedzieliśmy się jak bardzo elastyczna jest architektura RISCA V. Użytkownik może bez problemu dostosować rozmiar instrukcji, szerokości rejestrów itd. do swoich potrzeb. Atutem projektu okazała się jego otwartość, dzięki temu mogliśmy bez trudu znaleźć symulator, który pozwolił na oprogramowanie oraz symulowane uruchomienie programu napisanego w assemblerze dla RISC V. Jednak, jak wcześniej wspomniano, jest to młody projekt i niemożliwe okazało się znalezienie (dostępnego publicznie, darmowego) symulatora implementującego rozszerzenia wektorowe i jesteśmy przekonani, że w najbliższej przyszłości zostanie udostępniony przez producentów RISC V, symulator, który implementuje rozszerzenia wektorowe. Otwartość architektury prawdopodobnie wpłynie również na ceny tego procesora, ze względu na brak konieczności opłacania licencji za korzystanie z technologii.

Algorytm zaproponowany w projekcie, może być bezpośrednio implementowany w dostępnych na rynku procesorach RISC V, w przypadku szyfrowania Crystal Kyber dla wektorów o wielkości 32 bity, niemniej, zgodnie z naszą wiedzą istnieją procesory RISC V, które operują na znacznie większych wektorach i w najbliższej przyszłości szyfrowanie i deszyfrowanie tego typu procesorami będzie możliwe dla nawet najdłuższych kluczy w szyfrowaniu Crystal Kyber.

Procesory RISC V stosowane są głównie w zastosowaniach embedded, oczekuje się od nich małego zużycia energii. W związku z tym taktowane są stosunkowo niską częstotliwością. Zastosowanie przetwarzania wektorowego w implementacji Crystal Kyber pozwoli na szyfrowanie i dekodowanie poufnych informacji w czasie rzeczywistym, przy niewielkim zużyciu energii. Może to być wykorzystywane do bezpiecznej identyfikacji osób, zwierząt oraz zabezpieczeń przedmiotów.



Rysunek 11: Środowisko symulacyjne

7 Uwagi

Podczas implementacji szybkiego algorytmu NTT skorzystano z implementacji w języku JAVA, dostępnej na stronie [3]. Więcej informacji nt. procesora RISC V można znaleść w pracach [4], [5]. Praca powstała przy współpracy z Śp. Dr. hab. Jarosławem Pempera [6].

Literatura

[1] https://www.xilinx.com/products/boards-and-kits/alveo/u250.html/.

[2] https://sifive.cdn.prismic.io/sifive/7be0420e-dac1-4558-85bc-50c7a10787e7_p600_datasheet.pdf/.

[3] https://www.nayuki.io/page/number-theoretic-transform-integer-dft/.

[4] https://riscv.org/wp-content/uploads/2018/05/15.20-15.55-18.05.06.VEXT-bcn-v1.pdf/.

[5] https://inst.eecs.berkeley.edu/~cs152/sp20/handouts/sp20/riscv-v-spec.pdf/.

[6] https://pl.wikipedia.org/wiki/Jaros%C5%82aw_Pempera/.

[7] Carlos Aguilar-Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrède Lepoint. Nfllib: Ntt-based fast lattice library. In Kazue Sako, editor, *Topics in Cryptology - CT-RSA 2016*, pages 341–356, Cham, 2016. Springer International Publishing.

[8] Sedat Akleylek, Özgur Dağdelen, and Zaliha Yüce Tok. On the efficiency of polynomial multiplication for lattice-based cryptography on gpus using cuda. In Enes Pasalic and Lars R. Knudsen, editors, *Cryptography and Information Security in the Balkans*, pages 155–168, Cham, 2016. Springer International Publishing.

[9] Sedat Akleylek and Zaliha Yüce Tok. Efficient arithmetic for lattice-based cryptography on gpu using the cuda platform. In *2014 22nd Signal Processing and Communications Applications Conference (SIU)*, pages 854–857. IEEE, 2014.

[10] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. Math. Comp. 19, 297–30, 1965.

[11] Huimin Li, Nele Mentens, and Stjepan Picek. A scalable simd risc-v based processor with customized vector extensions for crystals-kyber. Cryptology ePrint Archive, Paper 2021/1648, 2021.

[12] Ayman W. Mohsen, Mohamed A. Sobh, and Ayman M. Bahaa-Eldin. Performance analysis of number theoretic transform for lattice-based cryptography. In *2018 13th International Conference on Computer Engineering and Systems (ICCES)*, pages 442–447, 2018.

[13] Lipeng Wan, Fangyu Zheng, Guang Fan, Rong Wei, Lili Gao, Yuewu Wang, Jingqiang Lin, and Jiankuo Dong. A novel high-performance implementation of crystals-kyber with ai accelerator. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors, *Computer Security – ESORICS 2022*, pages 514–534, Cham, 2022. Springer Nature Switzerland.

Dodatek

```
.data
V: .word 41, 2995, 26962, 292, 19169, 32391, 23281, 19718, 6334, 4827, 5705,
17421, 11478, 3902, 9961, 5447, 18467, 11942, 24464, 12382, 15724, 14604, 16827,
19895, 26500, 5436, 28145, 18716, 29358, 153, 491, 21726
PowerTab: .word 1, 30048, 4312, 11419, 6214, 31747,
22779, 482, 4740, 24017, 25646, 27729, 8433, 22854,
4061, 2050
_j: .word 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 0, 1, 4, 5
, 8, 9, 12, 13, 16, 17, 20, 21, 24, 25, 28, 29, 0, 1, 2, 3, 8, 9, 10, 11, 16, 17,
18, 19, 24, 25, 26, 27, 0, 1, 2, 3, 4, 5, 6,
7, 16, 17, 18, 19, 20, 21, 22, 23, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15
_jm: .word 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 2, 3, 6, 7,
10, 11, 14, 15, 18, 19, 22, 23, 26, 27, 30, 31, 4, 5, 6, 7, 12, 13, 14, 15, 20,
21, 22, 23, 28, 29, 30, 31, 8, 9, 10, 11, 12, 13, 14, 15, 24, 25, 26, 27, 28, 29,
30, 31, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31
_k: .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 0, 8, 0, 8,
0, 8, 0, 8, 0, 8, 0, 8,
0, 8, 0, 4, 8, 12, 0, 4, 8, 12, 0, 4, 8, 12, 0, 4, 8, 12, 0, 2, 4, 6, 8, 10,
12, 14, 0, 2, 4, 6, 8, 10, 12, 14, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15
v0: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
v1: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
v2: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
v3: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
v4: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
v5: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
v6: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
v7: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
mod: .word 32609
str1: .string "\n"
str2: .string " "
nby2: .word 16
n: .word 32

offset: .word 0
offset0: .word 0
.text
```

Rysunek 12: NTT w asemblerze Risc V - definicja danych

```
 ;a3 - vm dlugosc, a4 - rs1 adres src, a5 - vd adres dest, a6 - vs2 indeks
.macro vlxwu_v  vd, rs1,  vs2
        lw a3, nby2
        la a5, \vd
        la a4, \rs1
        la a6, \vs2
        addi t0, a4, 0
        jal ra, vlxwu_vf

.endm
;a3 - vm, a4 - rs1 adres src, a5 - vs3 adres dest, a6 - vs2 indeks
.macro vsxw_v  vs3, rs1,  vs2
        lw a3, nby2
        la a4, \rs1
        la a5, \vs3
        la a6, \vs2
        addi t0, a5, 0
        jal ra, vsxw_vf
.endm
;a3 - vm, a4 - vs2, a5 - vs1, a6 - vd
.macro vsub_vv  vd, vs2, vs1
        lw a3, nby2
        la a4, \vs2
        la a5, \vs1
        la a6, \vd
        jal ra, vsub_vvf
.endm
;a3 - vm, a4 - vs2, a5 - vs1, a6 - vd
.macro vadd_vv vd, vs2, vs1
        lw a3, nby2
        la a4, \vs2
        la a5, \vs1
        la a6, \vd
        jal ra, vadd_vvf
.endm
        ;a3 - vm dlugosc, a4 - vs2, rs1 / vd src/dest, a5 mod
.macro vremu_vx  vec, mod
        lw a3, nby2
        la a4, \vec
        lw a5, \mod
        jal ra, vremu_vxf
.endm
;a3 - vm dlugosc, a4 - vs1 src1, a5 - vs2 src2, a6 - vd dest
.macro vmul_vv  vd, vs2, vs1
        lw a3, nby2
        la a4, \vs1
        la a5, \vs2
        la a6, \vd
        jal ra, vmul_vvf
.endm
;a4 - vm dlugosc
;a0 a1 uzytkowe
.macro printResult V
        lw a3, n
        la a4, \V
        jal ra, printResultf
.endm
;a3 - vm dlugosc, a4 - G src, a5 - memory dest
;a0 a1 uzytkowe                          23
.macro vwritepoly  vd, memory, offset, len
        la a4, \memory
```

```
vlxwu_vf:
;a3 - vm dlugosc, a4 - rs1 adres src, a5 - vd adres dest, a6 - vs2 indeks
loop1:
        lw a0, 0(a6)
        addi t1, zero, 4
        mul t1, a0, t1
        add t1, t1, t0
        lw a0, 0(t1)
        sw a0, 0(a5)
        addi a5, a5, 4
        addi a3, a3, -1  ; Zdekrementuj zawartosc rejestru $t0 o 1
        addi a4, a4, 4
        addi a6, a6, 4
        bne a3, zero, loop1
ret

;a3 - vm, a4 - rs1 adres src, a5 - vs3 adres dest, a6 - vs2 indeks
vsxw_vf:
loop2:
        lw a0, 0(a6)
        addi t1, zero, 4
        mul a0, a0, t1
        add a5, t0 , a0
        sw a0, 0(a5)
        lw a0, 0(a4)
        sw a0, 0(a5)
        addi a3, a3, -1  ; Zdekrementuj zawartosc rejestru $t0 o 1
        addi a4, a4, 4
        addi a5, a5, 4
        addi a6, a6, 4
        bne a3, zero, loop2
ret

;a3 - vm, a4 - vs2, a5 - vs1, a6 - vd
vsub_vvf:
loop3:
        lw a0, 0(a4)
        lw a1, 0(a5)
        addi a4, a4, 4
        addi a5, a5, 4
        sub a0, a0, a1
        sw a0, 0(a6)
        addi a6, a6, 4
        addi a3, a3, -1  ; Zdekrementuj zawartosc rejestru $t0 o 1
        bne a3, zero, loop3
ret
;a3 - vm, a4 - vs2, a5 - vs1, a6 - vd
vadd_vvf:
loop4:
        lw a0, 0(a4)
        lw a1, 0(a5)
        addi a4, a4, 4
        addi a5, a5, 4
        add a0, a0, a1
        sw a0, 0(a6)
        addi a6, a6, 4
        addi a3, a3 ,-1  ; Zdekrementuj zawartosc rejestru $t0 o 1
        bne a3, zero, loop4
ret
```



```
        ;a3 - vm dlugosc, a4 - vs2, rs1 / vd src/dest, a5 mod
```

```
vremu_vxf:
loop5:
        lw a0, 0(a4)
        add a0, a0, a5
        rem a0, a0, a5
        sw a0, 0(a4)
        addi a4, a4, 4
        addi a3, a3, -1  ; Zdekrementuj zawartosc rejestru $t0 o 1
        bne a3, zero, loop5
ret

;a3 - vm dlugosc, a4 - vs1 src1, a5 - vs2 src2, a6 - vd dest
vmul_vvf:
loop6:
        lw a0, 0(a4)
        lw a1, 0(a5)
        addi a4, a4, 4
        addi a5, a5, 4
        mul a0, a0, a1
        sw a0, 0(a6)
        addi a6, a6, 4
        addi a3, a3, -1  ; Zdekrementuj zawartosc rejestru $t0 o 1
        bne a3, zero, loop6
ret


;a4 - vm dlugosc
;a0 a1 uzytkowe
printResultf:
loop7:
        lw a0, 0(a4)
        addi a4, a4, 4
        mv  a1, a0
        mv a0, a1
        li a7, 1
        ecall
        la a0, str2
        li a7, 4
        ecall
        addi a3, a3, -1  ; Zdekrementuj zawartosc rejestru $t0 o 1
        bne a3, zero, loop7
        la a0, str1
        li a7, 4
        ecall
ret


;a3 - vm dlugosc, a4 - G src, a5 - memory dest
;a0 a1 uzytkowe
vwritepolyf:
loop8:
        lw a0, 0(a4)
        addi a4, a4, 4
        sw a0, 0(a5)
        addi a5, a5, 4
        addi a3, a3, -1  ; Zdekrementuj zawartosc rejestru $t0 o 1
        bne a3, zero, loop8
ret
```


Rysunek 15: Funkcje realizujące operacje wektorowe - cd