THALES
Building a future we can all trust

Leveraging a Superscalar CVA6 to Implement NTT Instructions for PQC

Côme Allart, *Kévin Guilloux*, *Jean-Roch Coulon*, *André Sintzoff*, *Olivier Potin and Jean-Baptiste Rigaud*

Thales DIS & Mines Saint-Étienne

CASCADE

March 31st, 2026

INSPIRING INNOVATION SINCE 1816

Introduction

Context

Cryptography is used to **secure communications and transactions**

Quantum computers with enough qubits running Shor's algorithm will break asymmetric cryptography

Post-Quantum Cryptography (PQC): resist against quantum computer cryptanalysis

The National Institute of Standards and Technology (NIST) ratified in 2024

- FIPS 203: ML-KEM (Kyber) for key encapsulation and encryption
- FIPS 204: ML-DSA (Dilithium) for digital signature
- FIPS 205: SLH-DSA (SPHINCS+) as an alternative to ML-DSA

Application: **ML-DSA** digital signature scheme (Dilithium)

Introduction

The ML-DSA Digital Signature Scheme

ML-DSA consists of 3 algorithms

1. Key generation
2. Signature
3. Verification

ML-DSA consists of 3 algorithms

1. Key generation
2. Signature
3. Verification

All these algorithms contain 3 operation types

- Hash using Keccak
- Polynomial operations (multiplications via NTT)
- Other operations

ML-KEM uses the same operations

3 operation types

- Keccak (81 % for key generation according to PQ.V.ALU.E)
- Polynomial operations (12 %)
- Other operations (7 %)

3 operation types

- Keccak (81 % for key generation according to PQ.V.ALU.E)
- Polynomial operations (12 %)
- Other operations (7 %)

Most time-consuming operation: Keccak

- Best solution: accelerator outside the processor
- We can plug an existing accelerator to our processor

3 operation types

- **Accelerated** Keccak (81 $\rightarrow$ 3 %) for key generation according to PQ.V.ALU.E)
- Polynomial operations (12 $\rightarrow$ 63 %)
- Other operations (7 $\rightarrow$ 34 %)

Most time-consuming operation: Keccak

- Best solution: accelerator outside the processor
- We can plug an existing accelerator to our processor

Second most time-consuming operation: **polynomial multiplication via NTT**

- Both solutions are possible: accelerator outside the processor or inside the pipeline

How could NTT benefit from a superscalar processor pipeline?

Plan

1 Introduction

2 The CVA6 Superscalar Processor

3 NTT for Polynomial Multiplications

4 Model-Guided Methodology

5 Implementation Results

6 Conclusion and Perspectives

The CVA6 Superscalar Processor

1 Introduction

2 The CVA6 Superscalar Processor

3 NTT for Polynomial Multiplications

4 Model-Guided Methodology

5 Implementation Results

6 Conclusion and Perspectives

The CVA6 Superscalar Processor

- CVA6 is a RISC-V open-source in-order processor
- Created by F. Zaruba at ETH zürich (2019) and maintained by OPENHW FOUNDATION PROVEN PROCESSOR IP
- Completes execution out-of-order using a scoreboard
- Mature for industry

The CVA6 Superscalar Processor

- CVA6 is a RISC-V open-source in-order processor
- Created by F. Zaruba at ETH zürich (2019) and maintained by OPENHW FOUNDATION PROVEN PROCESSOR IP
- Completes execution out-of-order using a scoreboard
- Mature for industry

- Register-Transfer Level (RTL) model described in the SystemVerilog language
- More than 60 configuration options
  - 32- or 64-bit registers
  - Optionally supports various RISC-V extensions A, B, C, F, H, V, Zcb, Zcmp, Zicond, S and U modes
- Configurable performance features
  - Branch predictors
  - Scoreboard size
  - Dual-issue Superscalar
  - Etc.

The CVA6 Superscalar Processor

- CVA6 is a RISC-V open-source in-order processor
- Created by F. Zaruba at ETH zürich (2019) and maintained by OPENHW FOUNDATION PROVEN PROCESSOR IP
- Completes execution out-of-order using a scoreboard
- Mature for industry

- Register-Transfer Level (RTL) model described in the SystemVerilog language
- More than 60 configuration options
  - 32- or 64-bit registers
  - Optionally supports various RISC-V extensions A, B, C, F, H, V, Zcb, Zcmp, Zicond, S and U modes
- Configurable performance features
  - Branch predictors
  - Scoreboard size
  - **Dual-issue Superscalar**
  - Etc.

NTT for Polynomial Multiplications

1 Introduction

2 The CVA6 Superscalar Processor

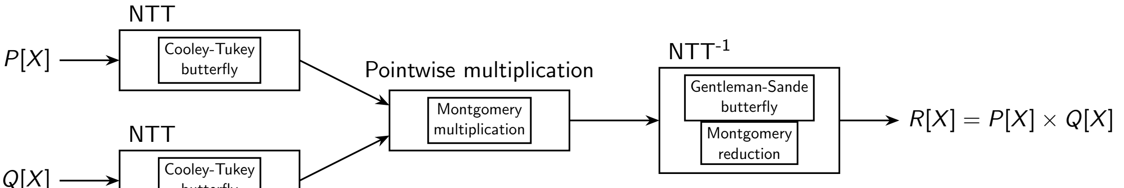
3 NTT for Polynomial Multiplications

4 Model-Guided Methodology

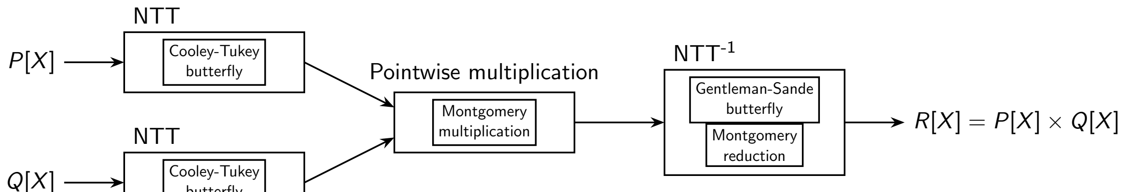
5 Implementation Results

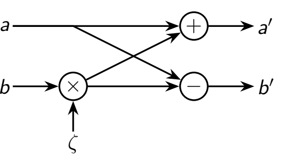
6 Conclusion and Perspectives

NTT for Polynomial Multiplications

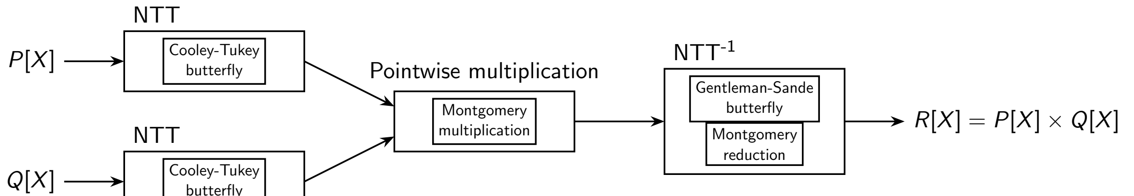


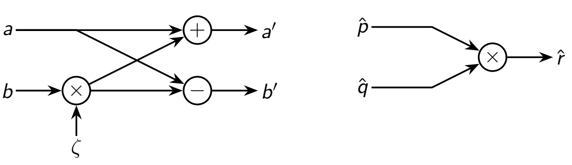
NTT for Polynomial Multiplications



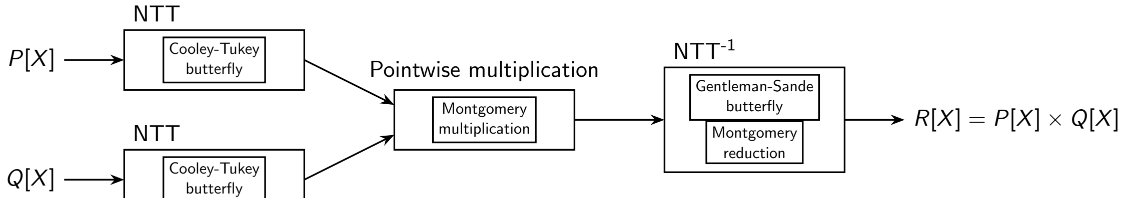


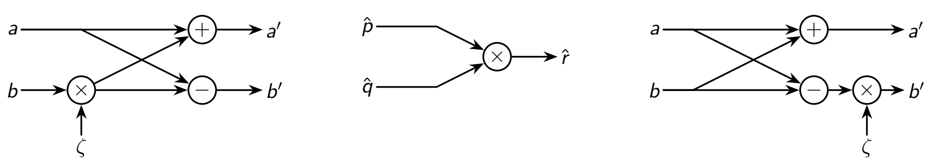
NTT for Polynomial Multiplications





NTT for Polynomial Multiplications





NTT for Polynomial Multiplications

NTT: $P[X]$ → Cooley-Tukey butterfly

NTT: $Q[X]$ → Cooley-Tukey butterfly

Pointwise multiplication: Montgomery multiplication

NTT^{-1}: Gentleman-Sande butterfly, Montgomery reduction → $R[X] = P[X] \times Q[X]$

Cooley-Tukey butterfly: inputs a, b, ζ; outputs a', b'

Pointwise multiplication: inputs $\hat{p}$, $\hat{q}$; output $\hat{r}$

Gentleman-Sande butterfly: inputs a, b, ζ; outputs a', b'

Butterfly operations have three inputs and two outputs

NTT for Polynomial Multiplications

State of the Art of NTT Hardware Acceleration

Duration of NTT, NTT^{-1} and pointwise multiplication (cycles)

Acceleration	Processor	Solution	NTT	NTT^{-1}	Pt.wise Mult.
Loosely-coupled Coprocessor	CVA6	CRYPHTOR	1074	1074	264 (1)
	CV32E40X	ATHOS	1531	1531	/
Inside the pipeline	RI5CY	32-bit RISC-V	17041	20372	4346
		PQ.V.ALU.E	**2705**	**3979**	**1274**
		Variation	-84 %	-80 %	-71 %
	Single-issue CVA6	64-bit RISC-V	38043	46266	/
		NTT Instructions	18554	21375	/
		Variation	-51 %	-54 %	/

Issue: how to handle the two output coefficients?

Model-Guided Methodology

1 Introduction

2 The CVA6 Superscalar Processor

3 NTT for Polynomial Multiplications

4 Model-Guided Methodology

5 Implementation Results

6 Conclusion and Perspectives

Model-Guided Methodology

Previous work

https://github.com/openhwgroup/cva6/tree/master/perf-model

APoG (*"apogee"*): Ariane Performance-only Guide

Model of CVA6 performance

https://github.com/openhwgroup/cva6/tree/master/perf-model

- Written in Python
- Matches **99%** of the 266,000 dynamic instructions of CoreMark

Model-Guided Methodology

Previous work

APoG (*"apogee"*): Ariane Performance-only Guide

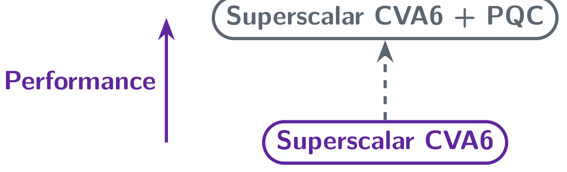
Model of CVA6 performance

https://github.com/openhwgroup/cva6/tree/master/perf-model

- Written in Python
- Matches **99%** of the 266,000 dynamic instructions of CoreMark
- Much easier to modify than RTL
- Guides the implementation of performance features
- Was **used to make CVA6 superscalar**

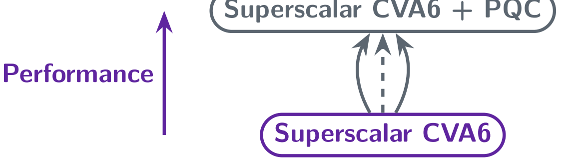
Côme Allart et al. *Using a Performance Model to Implement a Superscalar CVA6*, ACM CF '24, 2024

Model-Guided Methodology
Methodology



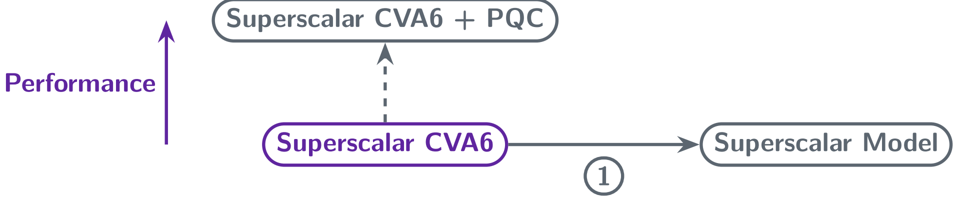
Model-Guided Methodology

Methodology



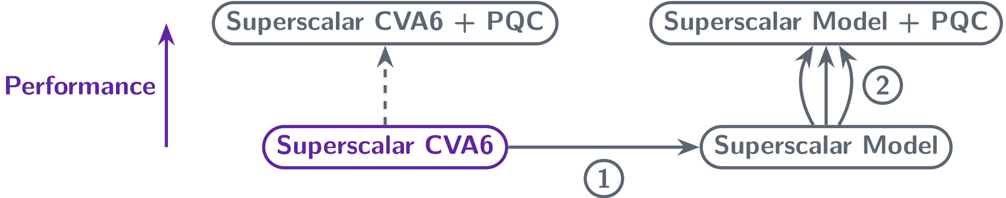
Model-Guided Methodology

Methodology



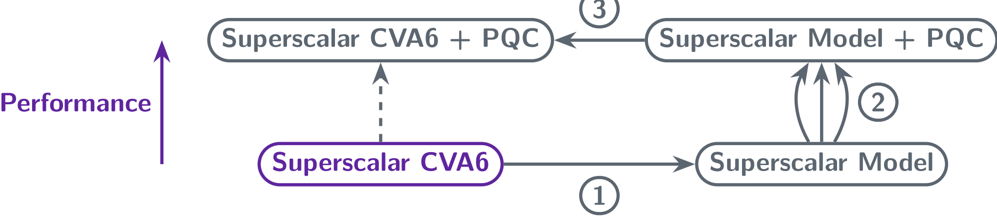
Model-Guided Methodology

Methodology



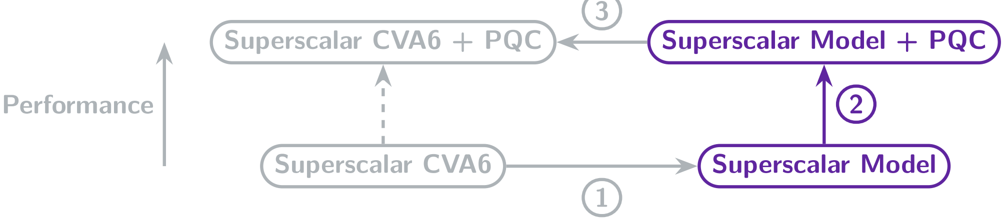
Model-Guided Methodology

Methodology



Model-Guided Methodology

Modeling Phase



Model-Guided Methodology

Modeling Phase

Operations to perform

- Load
- Butterfly
  - $a' = a + z.b \mod q$
  - $b' = a - z.b \mod q$
- Store

Model-Guided Methodology

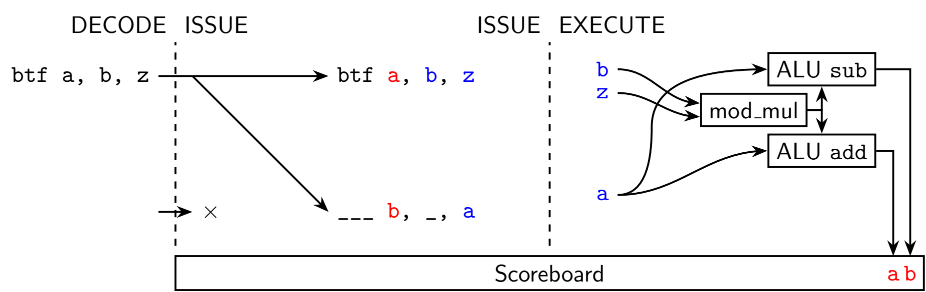
Modeling Phase

Operations to perform

- Load
- **Butterfly**
  - $a' = a + z.b \mod q$
  - $b' = a - z.b \mod q$
- Store

Instruction Modeling

- Prediction : $10335 \rightarrow$ **2719 cycles**



Model-Guided Methodology

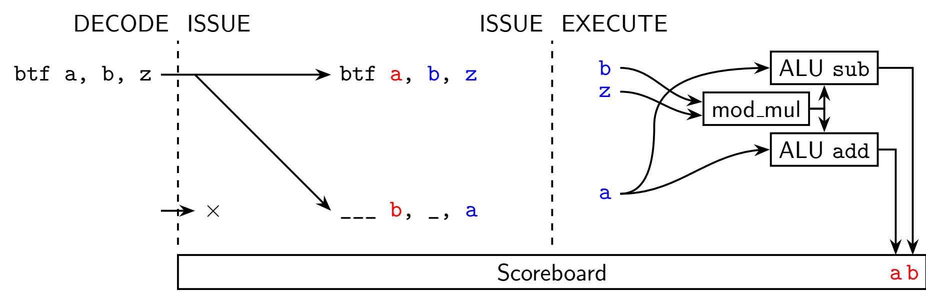
Modeling Phase

Operations to perform

- Load
- **Butterfly**
  - $a' = a + z.b \mod q$
  - $b' = a - z.b \mod q$
- Store

Instruction Modeling

- Assembly optimization
- Prediction : $10335 \rightarrow 2719 \rightarrow$ **2440 cycles**



Model-Guided Methodology

Modeling Phase

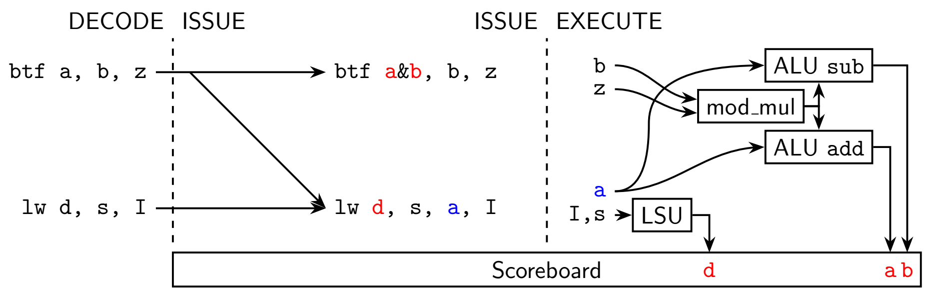
Operations to perform

- **Load**
- **Butterfly**
  - $a' = a + z.b \mod q$
  - $b' = a - z.b \mod q$
- Store

Instruction Modeling

- Assembly optimization
- Add load instructions in parallel

- Prediction : $10335 \rightarrow 2719 \rightarrow 2440 \rightarrow$ **2017 cycles**



Model-Guided Methodology

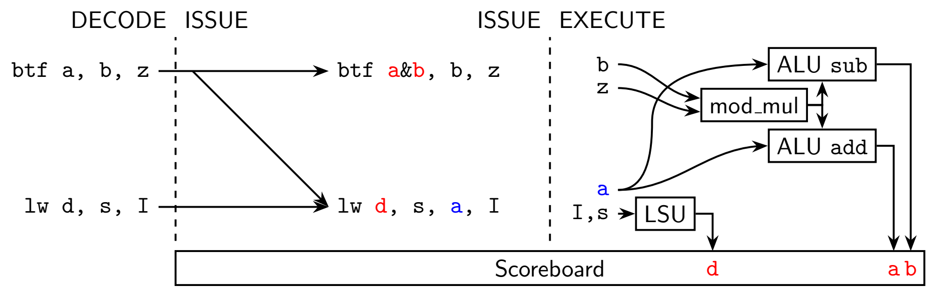
Modeling Phase

Operations to perform

- **Load**
- **Butterfly**
  - $a' = a + z.b \mod q$
  - $b' = a - z.b \mod q$
- Store

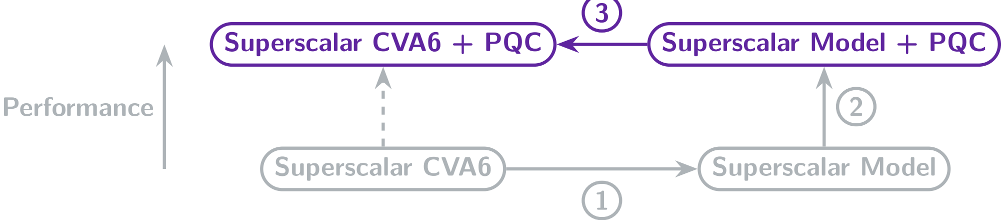
Instruction Modeling

- Assembly optimization
- Add load instructions in parallel
- Add triple-commit
- Prediction : 10335 → 2719 → 2440 → 2017 → **1747 cycles**



Model-Guided Methodology

Development phase



Model-Guided Methodology

Development phase

z
MUL
Reduction
ALU
btf.gs/mm
btf.ct
a
b
−
btf.ct/mm
btf.gs
−
a
b
btf.ct
btf.gs
+
btf.ct/gs
ALU2
base
offset
LSU
Scoreboard

a = data[1].rs2
b = data[0].rs1
z = data[0].rs2

base = data[1].rs1
offset = data[1].imm

before
added
added & critical path

Implementation Results

1 Introduction

2 The CVA6 Superscalar Processor

3 NTT for Polynomial Multiplications

4 Model-Guided Methodology

5 Implementation Results

6 Conclusion and Perspectives

Implementation Results

RTL Simulation Results

Duration of NTT, NTT^{-1} and pointwise multiplication (cycles)

Processor	Solution	NTT	NTT^{-1}	Pt.wise Mult.
Superscalar CVA6	32-bit RISC-V	**10335**	**11333**	**2188**
	Our solution	**1913**	**1892**	**807**
	Variation	**-81 %**	**-83 %**	**-63 %**
RI5CY	32-bit RISC-V	17041	20372	4346
	PQ.V.ALU.E	2705	3979	1274
	Variation	-84 %	-80 %	-71 %
Single-issue CVA6	64-bit RISC-V	38043	46266	/
	NTT Instructions	18554	21375	/
	Variation	-51 %	-54 %	/

Implementation Results

Hardware Implementation Results

Area

Hardware	Area (cost)	Maximum Frequency Reduction
Superscalar CVA6	109.4 $kGates_{eq}$ (reference)	(reference)
Our solution	114.1 $kGates_{eq}$ (**+4.7 $kGates_{eq}$**)	-2.8 %
RI5CY	43.7 $kGates_{eq}$ (reference)	(reference)
PQ.V.ALU.E	49.8 $kGates_{eq}$ (**+6.8 $kGates_{eq}$**)	Not on critical path

Power

Hardware	Avg. power consumption var.	Energy consumption var.
Superscalar CVA6	(reference)	(reference)
Our solution	**+21 %**	**-78 %**

Conclusion and Perspectives

1 Introduction

2 The CVA6 Superscalar Processor

3 NTT for Polynomial Multiplications

4 Model-Guided Methodology

5 Implementation Results

6 Conclusion and Perspectives

Conclusion and Perspectives

Goal: **Improve performance of the superscalar CVA6 processor for PQC**

Model-guided method to evaluate solutions before hardware development

Result: Butterfly instructions allow **executing NTT 5 times faster** than base superscalar CVA6
$\Rightarrow$ **faster than similar implementations based on single-issue processors**.

Perspectives:

- Add instructions for ML-KEM, with the appropriate q value
- Perform side-channel and fault injection evaluations of CVA6
- Secure cryptographic instructions against hardware attacks

Acknowledgments

All this work is based on the open-source CVA6 processor and model

https://github.com/openhwgroup/cva6

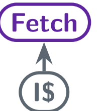
These activities are supported by the ISOLDE project funded by the Key Digital Technologies Joint Undertaking (KDT JU) under grant agreements 101112274. The present action reflects only the authors' view; the European Commission and the JU are not responsible for any use that may be made of the information it contains.

CVA6 Pipeline

- 6-stage pipeline

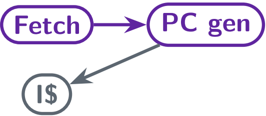
CVA6 Pipeline

- 6-stage pipeline



CVA6 Pipeline

- 6-stage pipeline

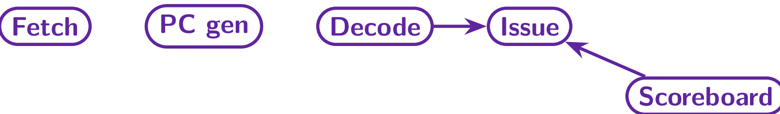


CVA6 Pipeline

- 6-stage pipeline

CVA6 Pipeline

- 6-stage pipeline
- **In-order operand reading**



CVA6 Pipeline

- 6-stage pipeline
- **In-order operand reading**

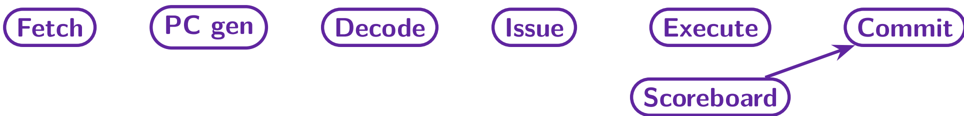
CVA6 Pipeline

- 6-stage pipeline
- In-order operand reading
- **Out-of-order execution**

Fetch | PC gen | Decode | Issue | Execute → Scoreboard

CVA6 Pipeline

- 6-stage pipeline
- In-order operand reading
- Out-of-order execution



CVA6 Pipeline

- 6-stage pipeline
- In-order operand reading
- Out-of-order execution
- **Dual-issue Superscalar**

PC gen

Decode

Issue

Execute

Commit

Scoreboard

16-coefficient NTT Example

$A[X]$

a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}

$\hat{a}_0$, $\hat{a}_1$, $\hat{a}_2$, $\hat{a}_3$, $\hat{a}_4$, $\hat{a}_5$, $\hat{a}_6$, $\hat{a}_7$, $\hat{a}_8$, $\hat{a}_9$, $\hat{a}_{10}$, $\hat{a}_{11}$, $\hat{a}_{12}$, $\hat{a}_{13}$, $\hat{a}_{14}$, $\hat{a}_{15}$

$\hat{A}[X]$

normal field | layer 1 | layer 2 | layer 3 | layer 4 | NTT field

Polynomial Matrix-Vector Multiplication

Details of Matrix-Vector Multiplication As (ML-DSA-65 = Dilithium 3 parameters):

- $A = (A_{ij})_{(i,j)\in[0;6[\times[0;5[}$
- $s = (s_j)_{j\in[0;5[}$
- $As = ((As)_i)_{i\in[0;6[}$
- $(As)_i = \sum_j (A_{ij} \times s_j)$

Where $\times$ is a 256-coefficient polynomial multiplication:

- $P[X] = \sum_i p_i X^i$
- $Q[X] = \sum_j q_j X^j$
- $P[X] \times Q[X] = \sum_i (\sum_j p_i q_j X^{i+j})$

$O(n^2)$ complexity

NTT Multiplication Complexity

In the NTT field, multiplication is simpler: $\widehat{P}[X] \times \widehat{Q}[X] = \sum_i \widehat{p}_i \widehat{q}_i X^i$
$O(n)$ complexity

However, NTT and NTT^{-1} transforms are required

- $\widehat{A}$ is already sampled in the NTT field
- $\mathbf{NTT}^{-1}(\widehat{A} \circ \mathbf{NTT}(s))$

$O(n \log n)$ complexity

Each polynomial needs:

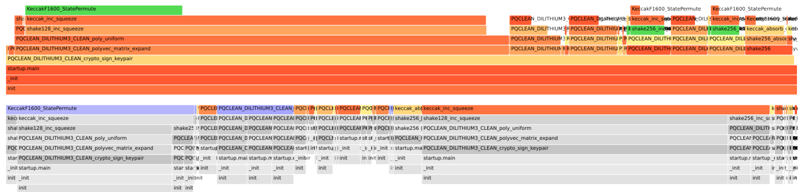
- $\widehat{s}_i = NTT(s_i)$ before multiplication
- $(As)_i = NTT^{-1}(\widehat{As_i}))$ after multiplication

$O(n \log n)$ **complexity, instead of** $O(n^2)$ **without NTT**

ML-DSA Key Generation

Our Profiling Results

PQClean : open-source software implementation



- Shake = 80%
- NTT et NTT^{-1} = 11%

Performance gain from the superscalar feature:

- Single issue : 4 468 886 cycles
- Superscalar : 3 003 757 cycles (-33%)