A Fast Open-Source Extended GCD Accelerator

Kavya Sreedhar, Mark Horowitz, Christopher Torng
Stanford University
skavya@stanford.edu

June 18, 2023

Cryptography relies on hard problems

- Modern cryptography is based on computationally hard problems
  - Typically require large-integer arithmetic
- Execution time of computation for these problems is critical

Many hard problems rely on extended GCD

XGCD computes Bézout coefficients $\boldsymbol{b_a}, \boldsymbol{b_b}$ satisfying Bézout's Identity

$$\boldsymbol{b_a}, \boldsymbol{b_b} \ : \ \boldsymbol{b_a} * a_0 + \boldsymbol{b_b} * b_0 = \gcd(a_0, b_0)$$

There is an increasing need for faster XGCD

2018: Verifiable delay functions [1]

- Useful for consensus protocols
- Can be efficiently verified
- Require fixed time for evaluation

[1] Boneh et al. Verifiable delay functions. Crypto 2018.

There is an increasing need for faster XGCD

2018: Verifiable delay functions [1]

- Useful for consensus protocols
- Can be efficiently verified
- Require fixed time for evaluation

2021: XGCD found to be fastest way to compute modular inverses [2]

- Used widely in cryptography
- Find $x^{-1} : x * x^{-1} = 1 \ (mod\ p)$
  - Since x is secret, this operation needs to be constant-time

[1] Boneh et al. Verifiable delay functions. Crypto 2018. **[2]** Bernstein and Yang. Fast constant-time gcd computation and modular inversion. CHES 2019.

There is an increasing need for faster XGCD

2018: Verifiable delay functions [1]

- Useful for consensus protocols
- Can be efficiently verified
- Require fixed time for evaluation

- XGCD takes 91% of execution time

1024-bits, not constant-time

2021: XGCD found to be fastest way to compute modular inverses [2]

- Used widely in cryptography
- Find $x^{-1}: x * x^{-1} = 1 \ (mod\ p)$
  - Since x is secret, this operation needs to be constant-time

- XGCD takes 100% of execution time

255-bits, constant-time

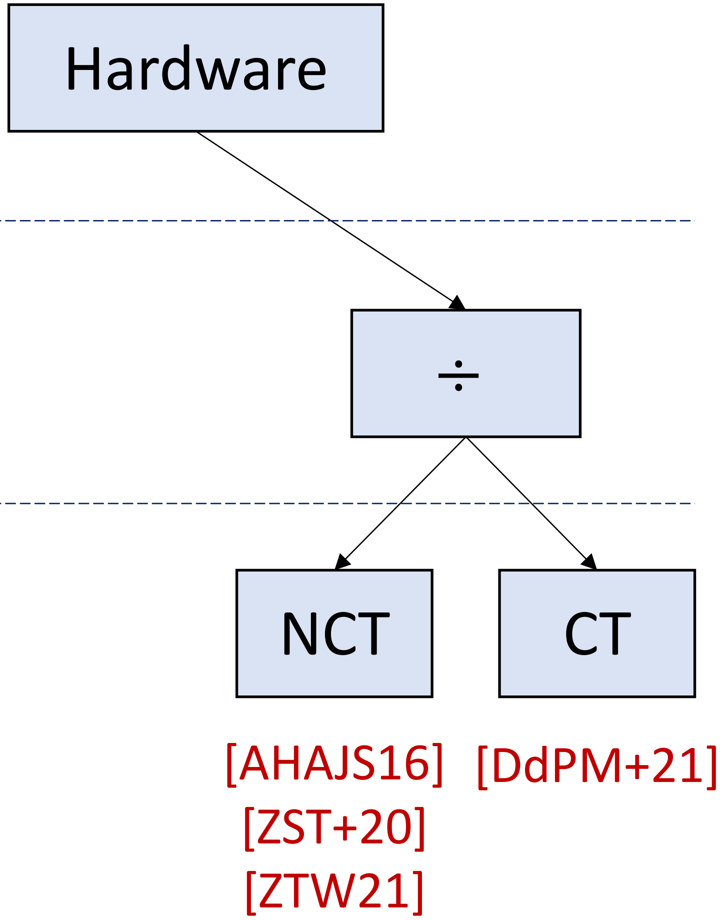
[1] Boneh et al. Verifiable delay functions. Crypto 2018. **[2]** Bernstein and Yang. Fast constant-time gcd computation and modular inversion. CHES 2019.

Previous view of XGCD design space

Target Platform

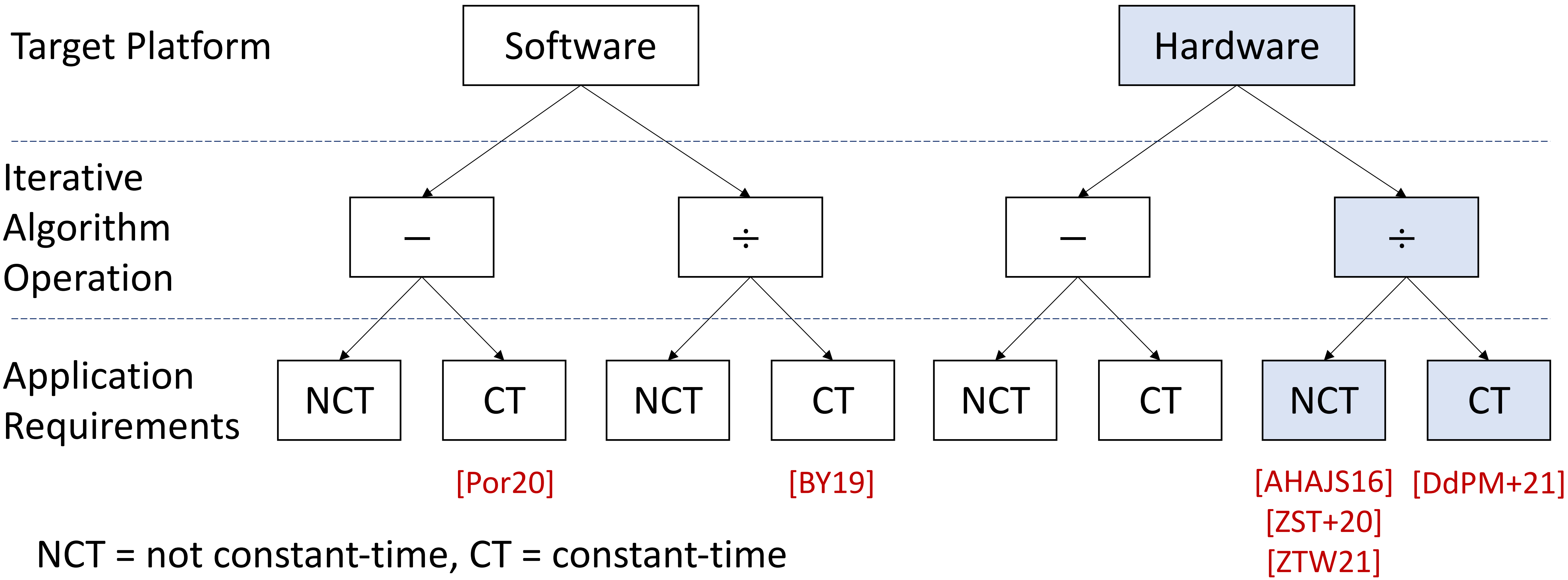
Iterative
Algorithm
Operation

Application
Requirements



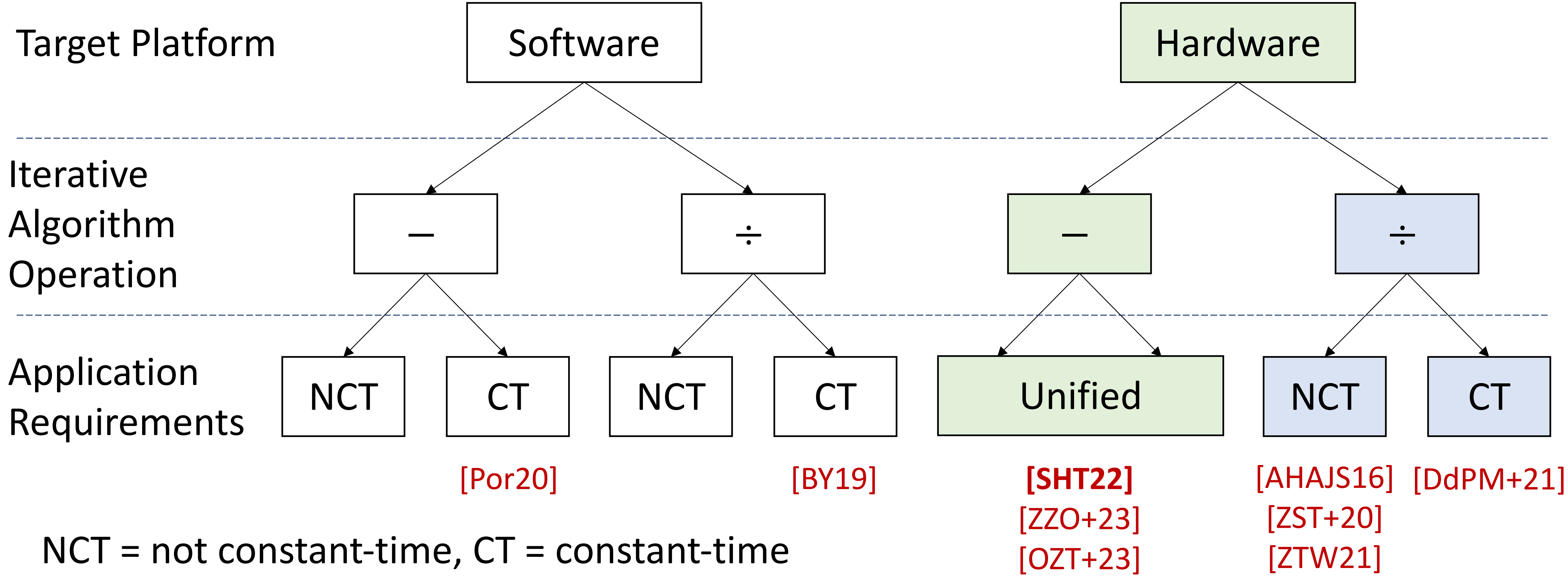
NCT = not constant-time, CT = constant-time

We explore the broader XGCD design space



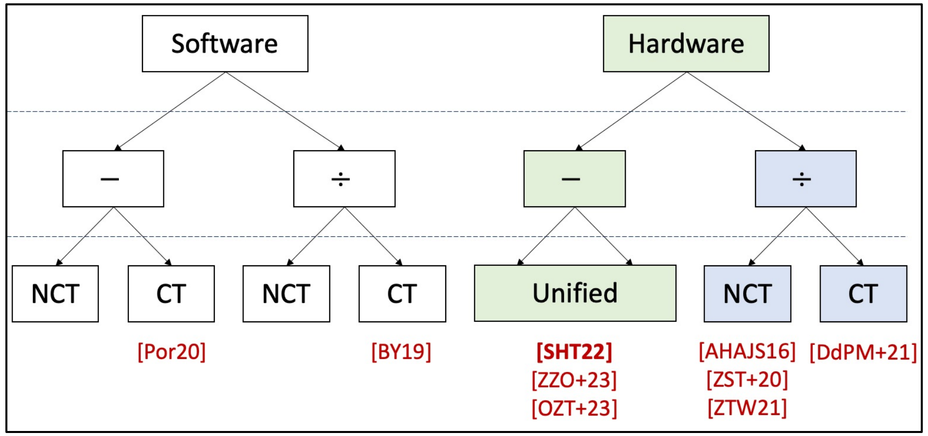
NCT = not constant-time, CT = constant-time

We explore the broader XGCD design space

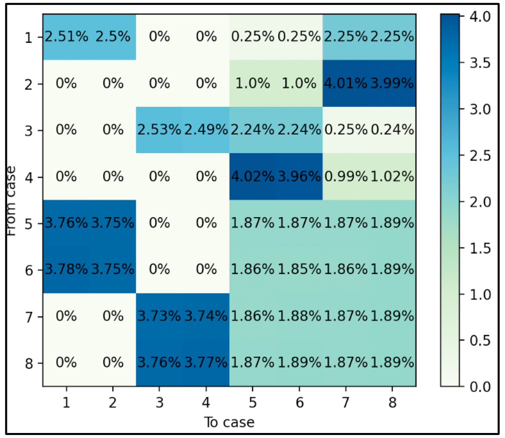


NCT = not constant-time, CT = constant-time

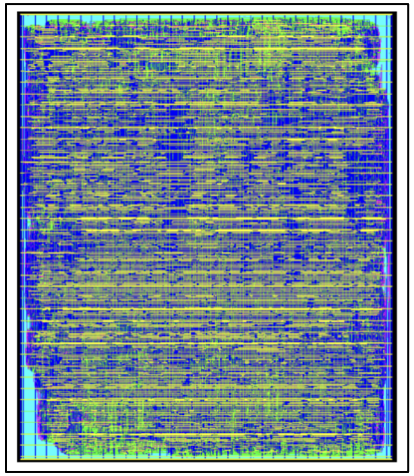
Outline



Design Space



Performance Case Studies



Open-Source Accelerator

Hardware allows for short iteration times

Target Platform	**Software**	**vs**	**Hardware**
Number of Iterations	From algorithm		From algorithm
Constrained to ISA	Yes		No

Execution time = number of iterations $*$ iteration time

The control over iteration time in hardware opens the opportunity to accelerate simpler algorithms that require more iterations.

Subtraction-based algorithms are faster

	Stein [Ste67]		Euclid (300 BC)
Iterative Algorithm Operation	$\boxed{-}$	**vs**	$\boxed{\div}$
GCD-preserving Transformation	$\gcd(a,b) = \gcd(a-b,b)$		$\gcd(a,b) = \gcd(a \bmod b, b)$

Also divides by 2 if possible

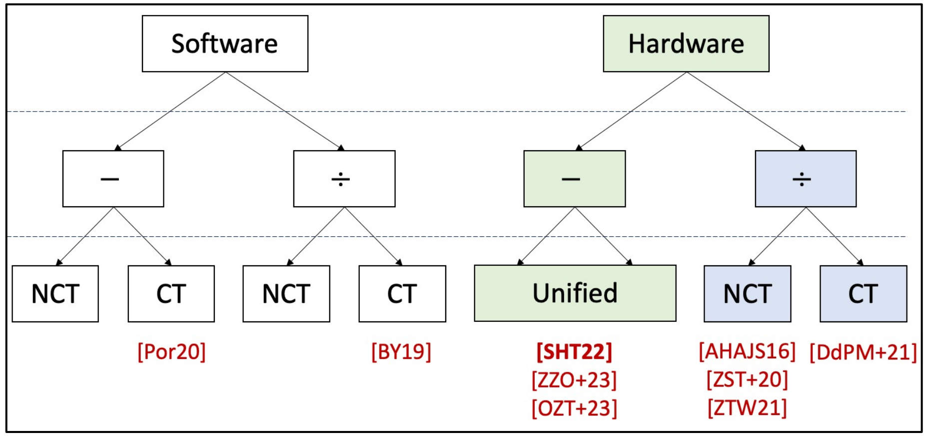
Execution time = number of iterations $*$ iteration time

Subtraction-based algorithms result in short critical paths and reduce overall latency compared to **division-based algorithms**.

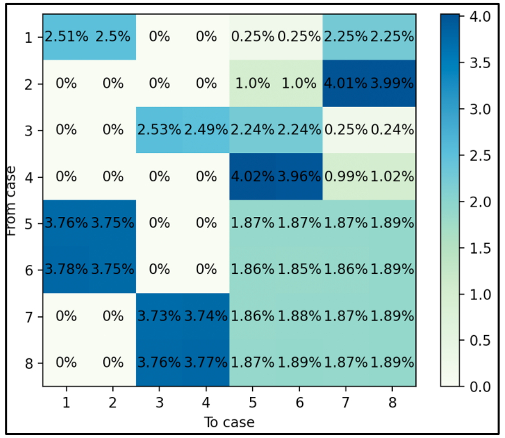
Our unified design with constant-time config

Application Requirements	**Not constant-time**	vs	**Constant-time**
Approach	Reduce inputs until GCD		Pad to worst-case cycle count
Termination Condition	$a == 0$ or $b == 0$		Cycle count equal to worst case

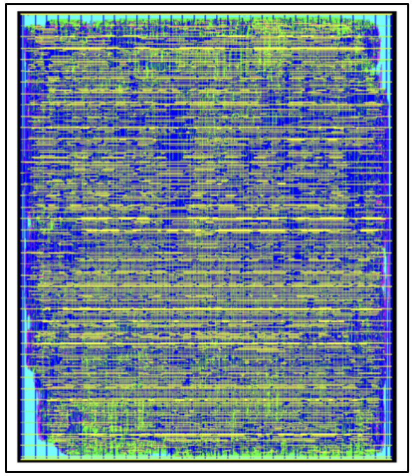
Outline



Design Space



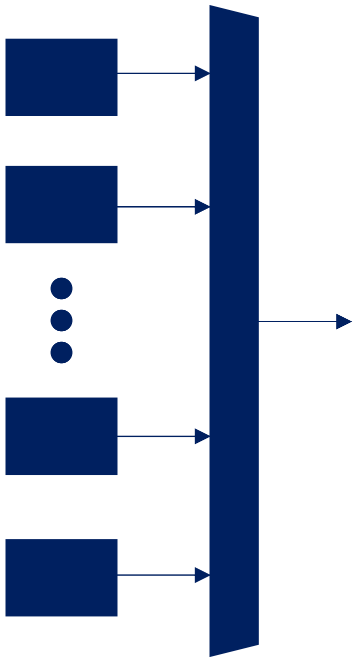
Performance Case Studies



Open-Source Accelerator

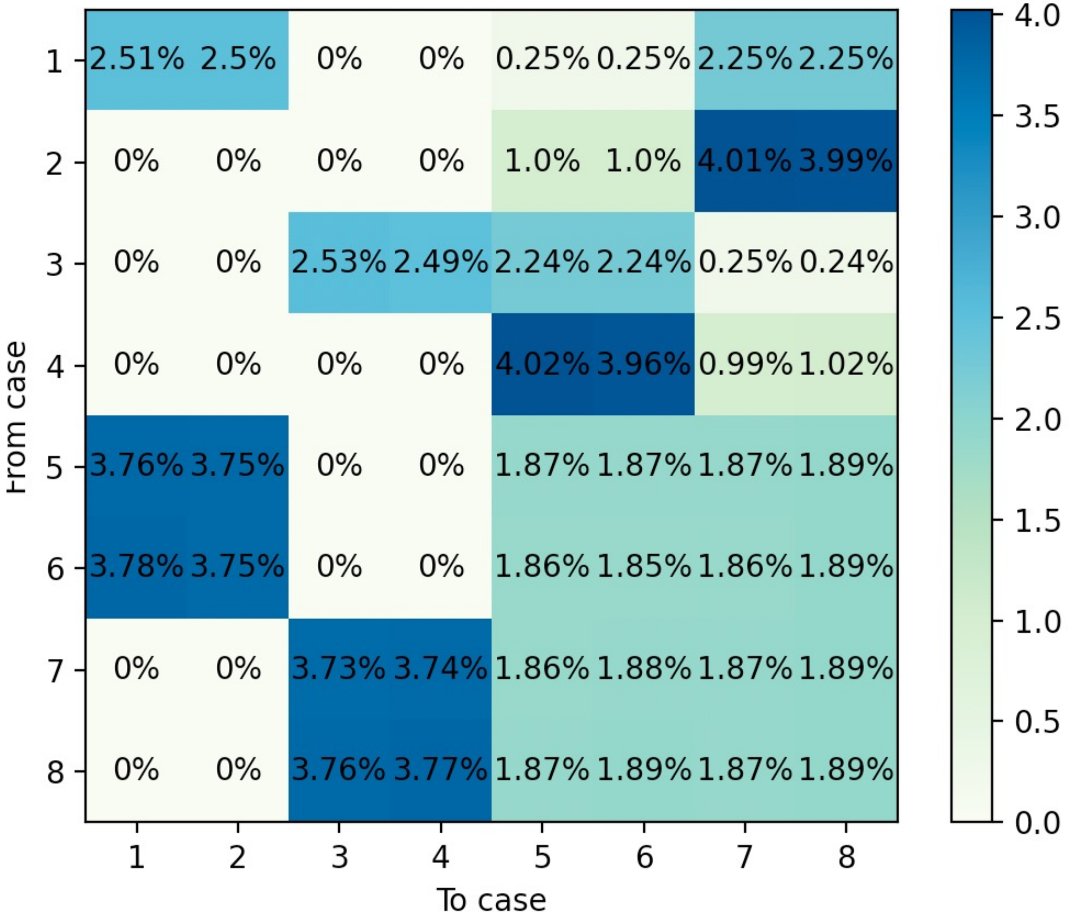
Case study #1: composing operations

- Design selects across many parallel operations
- Some operations are compositions of others
- In an iterative algorithm, composing reduces cycles
- When should we stop composing?



22 parallel operations

Transition matrix



- Case 1: $a = \frac{a}{4}$
- Case 2: $a = \frac{a}{2}$
- Case 3: $b = \frac{b}{4}$
- Case 4: $b = \frac{b}{2}$
- Case 5: $a = \frac{a+b}{4}$
- Case 6: $a = \frac{b-a}{4}$
- Case 7: $b = \frac{a+b}{4}$
- Case 8: $b = \frac{b-a}{4}$

Random 1024-bit inputs

When should we stop composing?

- Not constant-time
  - Stop when critical path delay increase exceeds cycles decrease
  - When a or b is even: Divide by up to 8
  - When a and b are odd: Divide by 4

When should we stop composing?

- Not constant-time
  - Stop when critical path delay increase exceeds cycles decrease
  - When a or b is even: Divide by up to 8
  - When a and b are odd: Divide by 4

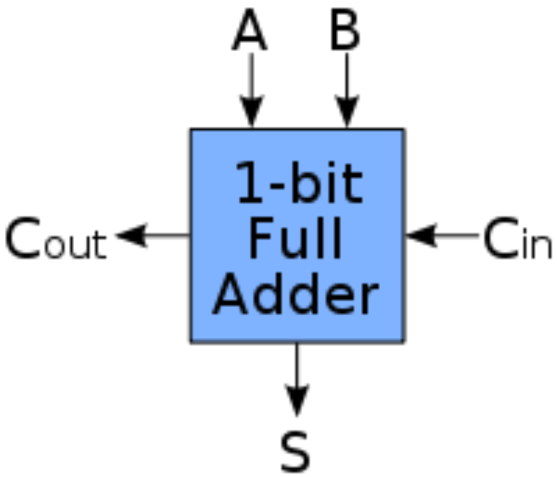
- Constant-time
  - Stop when transitions are not guaranteed
  - When a or b is even: Divide by 2
  - When a and b are odd: Divide by 4

Case study #2 relies on carry-save adders

- The fastest adder is a carry-save adder (CSA)
  - Eliminates carry propagation, requiring $O(1)$ delay
  - Stores numbers in CSA form or redundant binary form

```
 1 1 1 0
   1 1 0 1 (a)
+  1 1 1 0 (b)
   0 0 1 0 (c)
 ---------------
 1 1 1 0 1
```

```
   1 1 0 1 (a)
+  1 1 1 0 (b)
   0 0 1 0 (c)
 -------------------
+  0 0 0 1 sum
 1 1 1 0 0 carry
 -------------------
 1 1 1 0 1
```



Case study #2: fast termination detection

- Design terminates when a or b is equal to 0
- The values of a and b are not directly known in CSA form

carry	0000	0001	1011
sum	0000	1111	0101

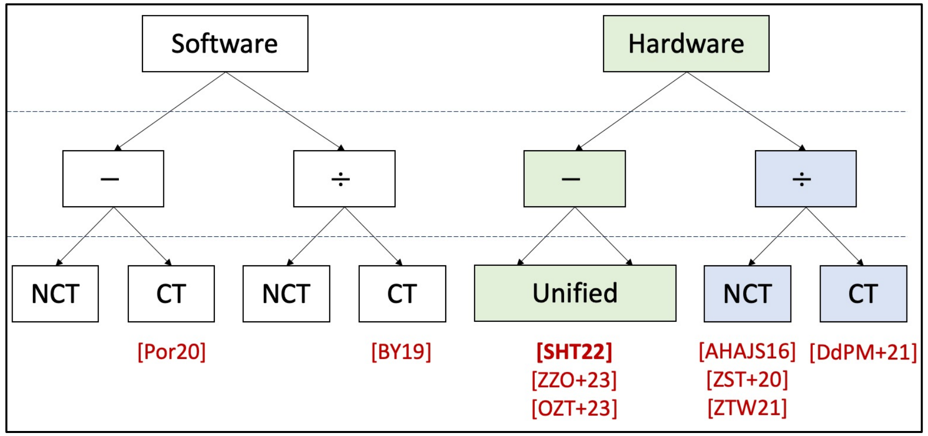
- Recovering a, b requires long carry propagation
- How can we compare a and b to 0 efficiently every iteration?

How can we compare a and b to 0?

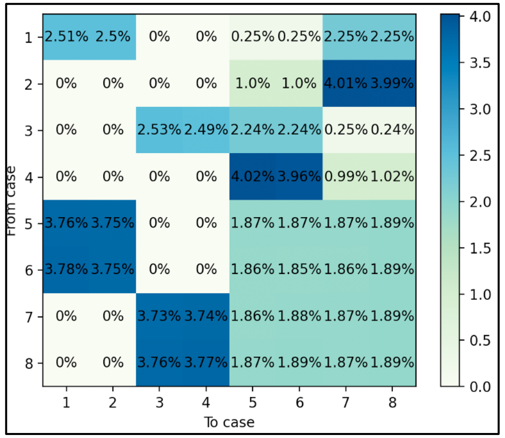
- Can shorten carry propagation by tracking $\alpha \approx \log_2 a, \beta \approx \log_2 b$
- However, α, β can diverge from $\log_2 a, \log_2 b$
- Can occasionally correct α, β to be the true values of $\log_2 a, \log_2 b$

Correction Frequency	Average Added Cycle Overhead
16	0.5%
64	2.0%
256	7.3%
Never	13%

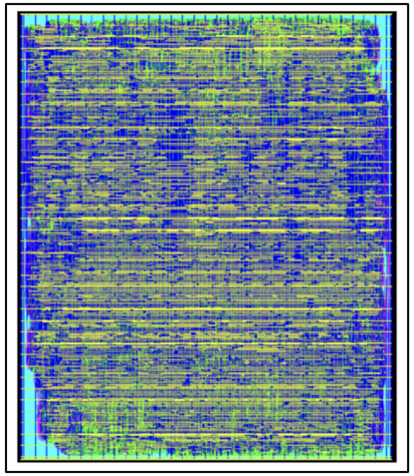
Outline



Design
Space

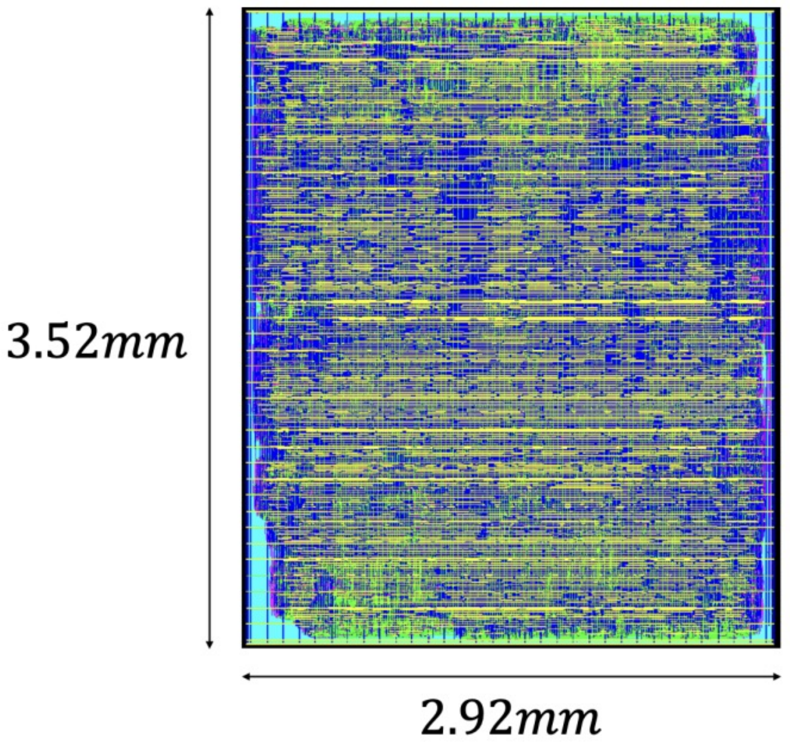


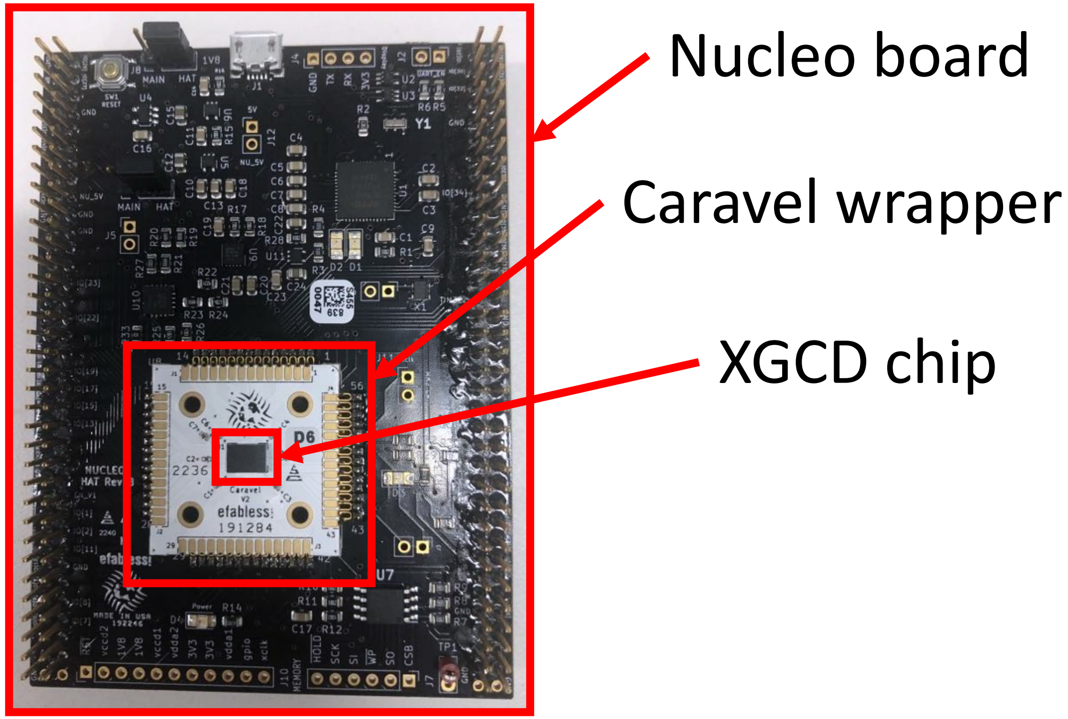
Performance
Case Studies



Open-Source
Accelerator

Open-Source accelerator with SKY130





Fabricated with the Efabless Open MPW2 Shuttle, sponsored by Google

https://efabless.com/open_shuttle_program

Area is dominated by Bézout variable updates

Module	Area (mm^2)	% of Area
Initial computation	0.27	4.8
a, b update (2-count)	0.31	5.5
Bézout coefficient update (4-count)	3.84	68
Control variable updates	0.57	10
Final result calculation	0.36	6.4
JTAG for Chip IO	0.22	3.6
Miscellaneous	0.10	1.7
Total	5.66	100

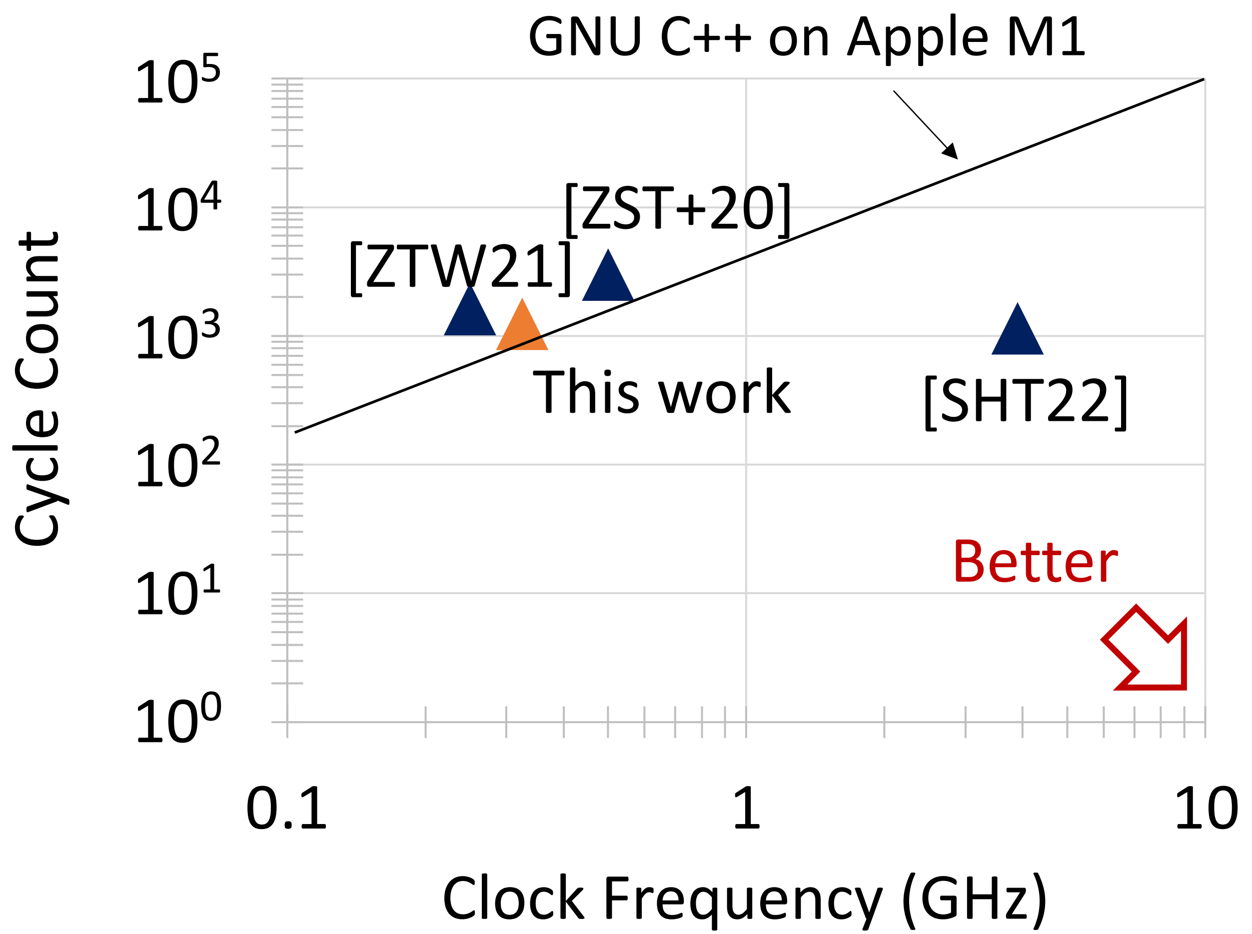
Critical path for ASIC in SKY130

Operation	Delay (ns)
DFF CLK to Q	0.54
CSA 1	0.51
CSA 2	0.67
CSA 3	0.56
Shift in CSA form	0.22
Late select multiplexers	0.30
Precomputing control	0.14
Library setup time	0.07
Total	3

Carry-save adder logic

Control overhead

Related work comparison



<u>Our ASIC simulation</u>

- 38X faster than software
- 14X faster than state-of-the-art ASIC

* Graph shows absolute times in us

* Comparisons are with all prior work technology-scaled to 180-130nm

Putting the open-source in OSCAR

RTL and SKY130 physical design files

https://github.com/kavyasreedhar/sreedhar-xgcd-hardware-ches2022

Efabless Caravel user project integration for MPW2 tapeout

https://github.com/kavyasreedhar/caravel_user_project

Takeaways

- Recent advanced cryptography developments heavily rely on fast XGCD
- Iterative subtraction and carry-save arithmetic enable high performance
- Open-source XGCD accelerator demonstrated in SKY130nm