Fast Extended GCD for Large Integers for Verifiable Delay Functions

Kavya Sreedhar

AHA Meeting 8.5.2021

Advanced cryptography needs faster GCD

- GCD is a well-established operation in number theory with
  - Wide-ranging applications in cryptography
  - A focus on developing fast algorithms in the 1980s/90s

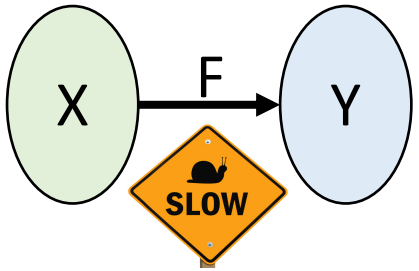
- Verifiable delay functions (VDFs), introduced in 2018, create a new need for understanding how fast work can be in this space

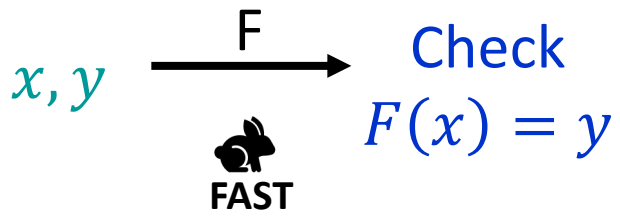
Our goal: build fast GCD for the VDF application domain

Verifiable delay functions

Verifiable delay function (VDF)

- Function that is slow to compute but easy to verify
- Defined by 3 algorithms: *setup, evaluate, verify*
  - *Evaluate* takes a fixed number of sequential steps T
  - *Verify* must be efficient when checking the result of *evaluate*





Delay: Requires minimum evaluation time despite any available parallelism

Verifiable: Can be efficiently verified

The crypto community is excited about VDFs

Chia Network Announces 2nd VDF Competition with $100,000 in Total Prize Money

Matt Howard and Bram Cohen — April 4, 2019

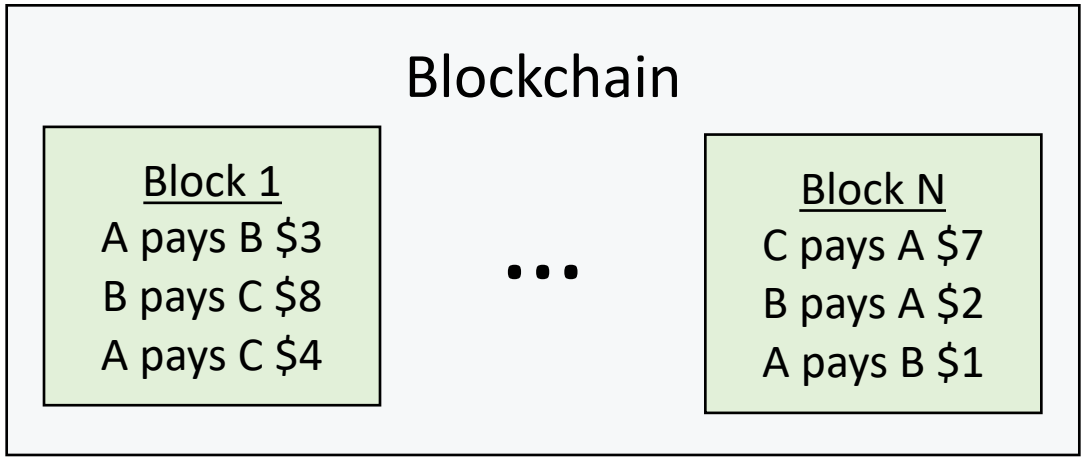
THOMAS SIMMS APR 22, 2019

Protocol Labs and Ethereum Foundation Team Up to Research Verifiable Delay Functions

Updated Aug 11, 2019 at 6:57 p.m. PDT

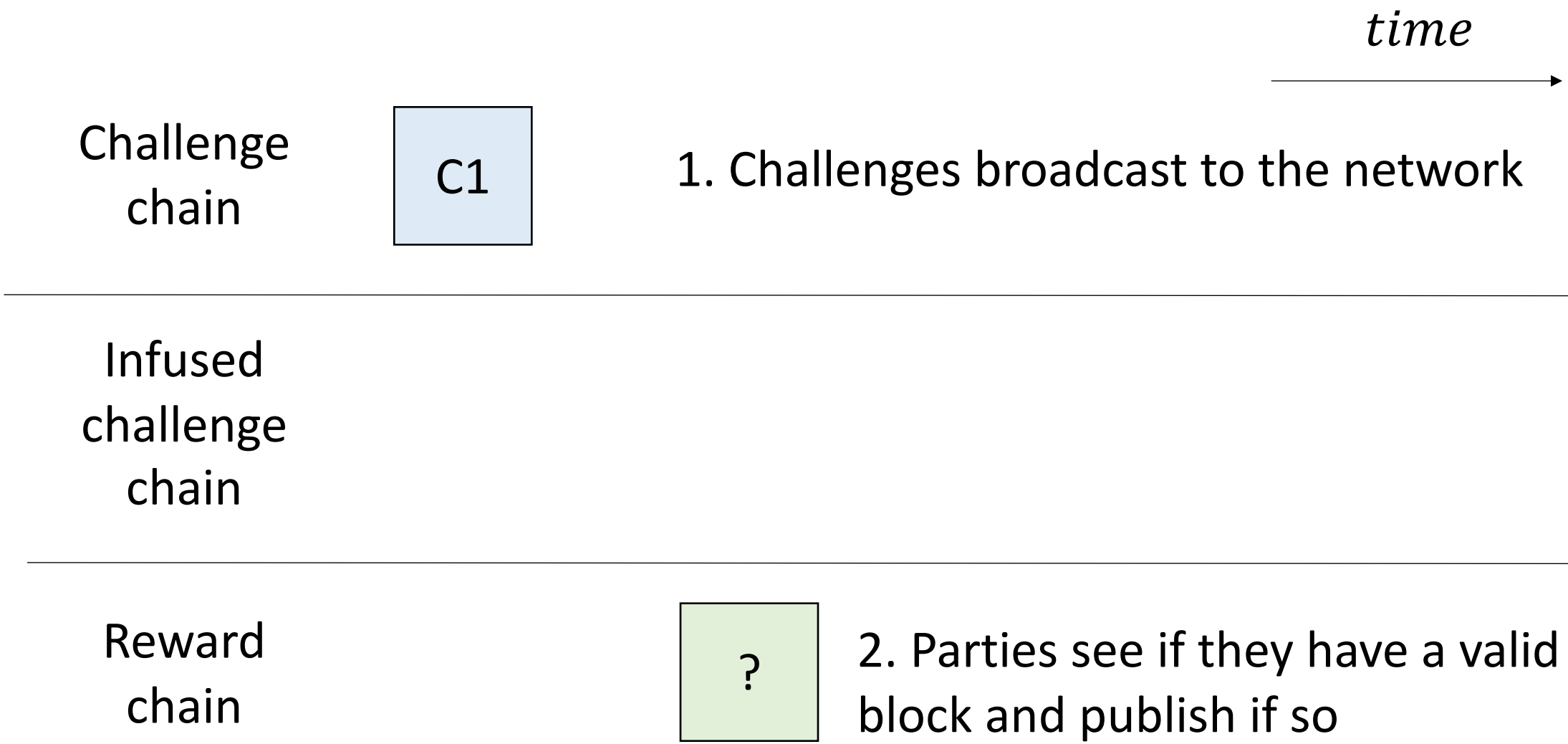
At the cutting edge of blockchain research is a potential $15 million dollar venture by the Ethereum Foundation centered around a technology called Verifiable Delay Functions (VDFs).

VDFs can secure blockchain systems

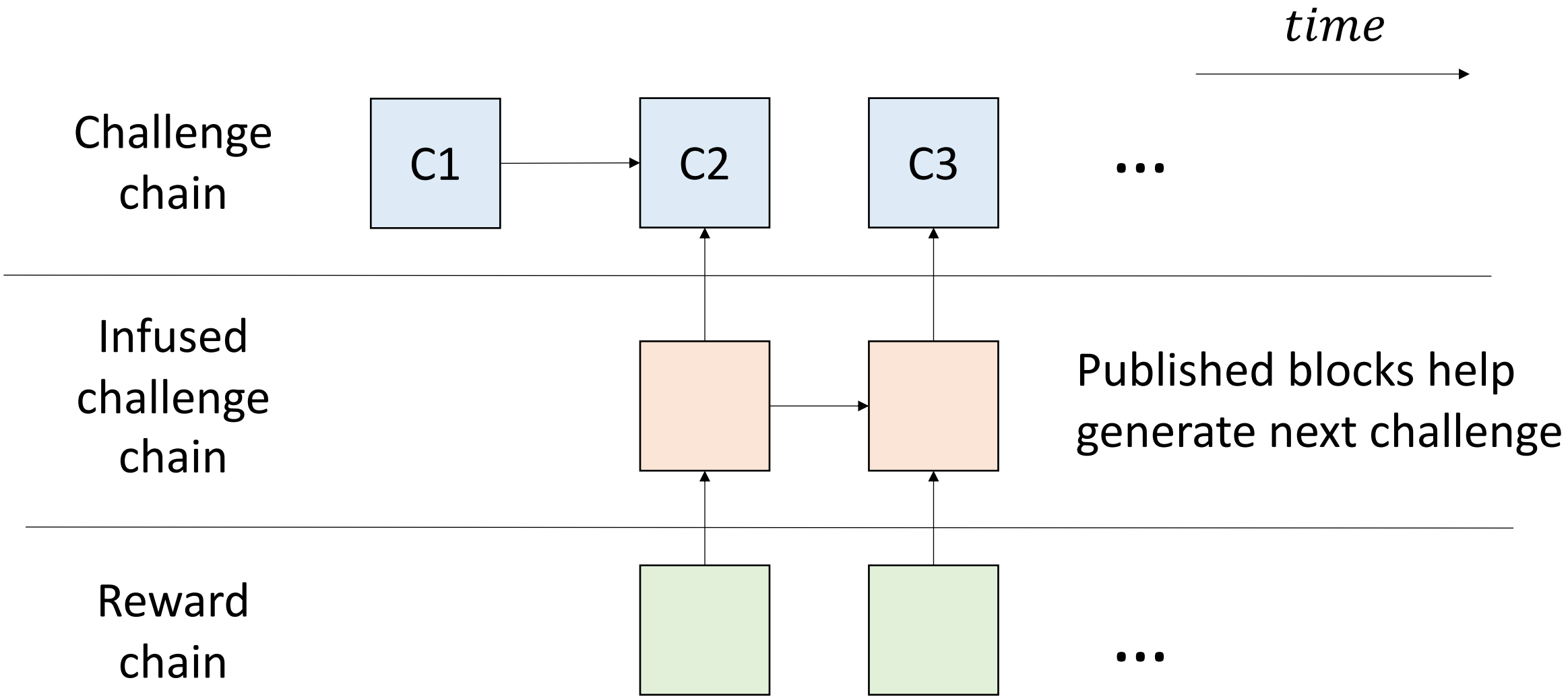


- Transactions are recognized once validated and added to the chain
  - Verifiers validate transactions (and usually receive a reward for doing so)
  - Secure blockchains prevent attackers from taking over a majority of validation

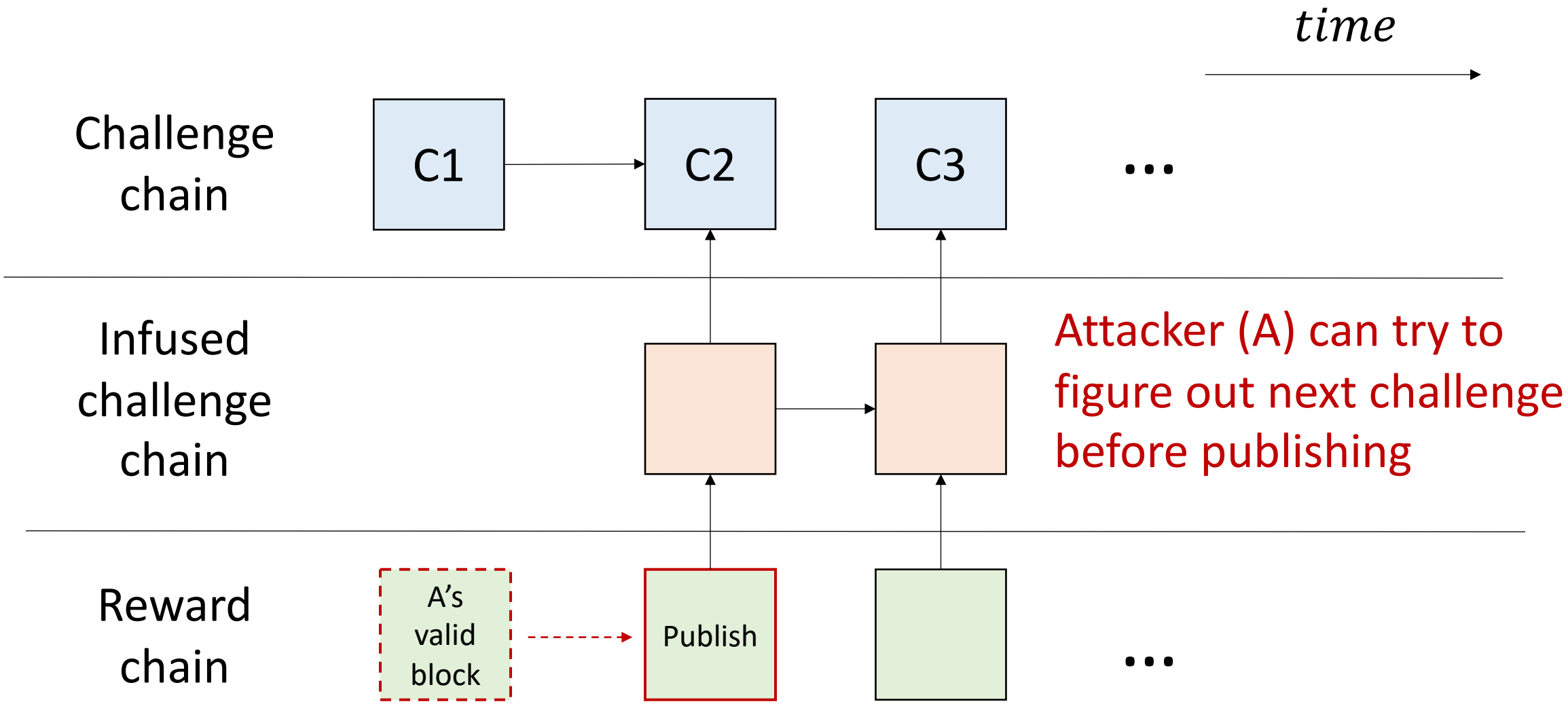
VDFs can secure blockchain systems



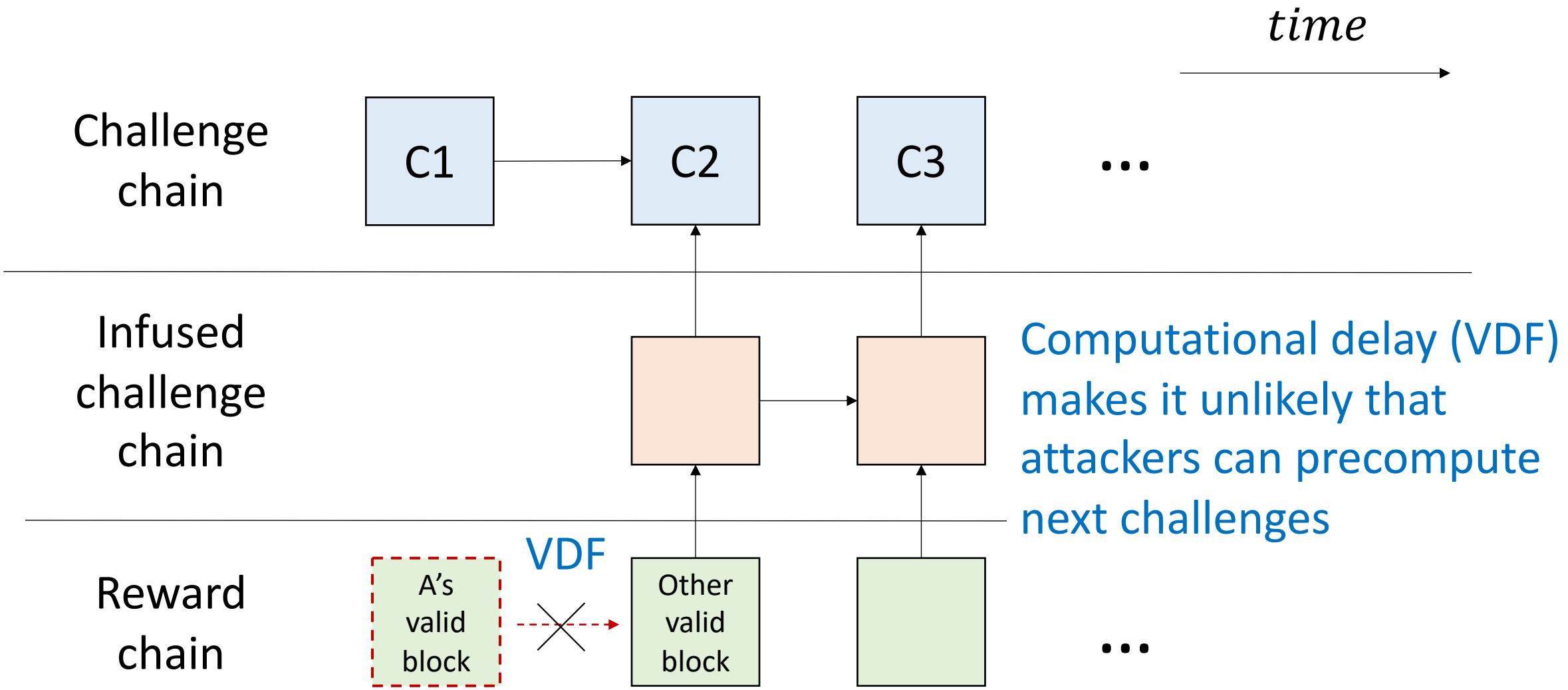
VDFs can secure blockchain systems



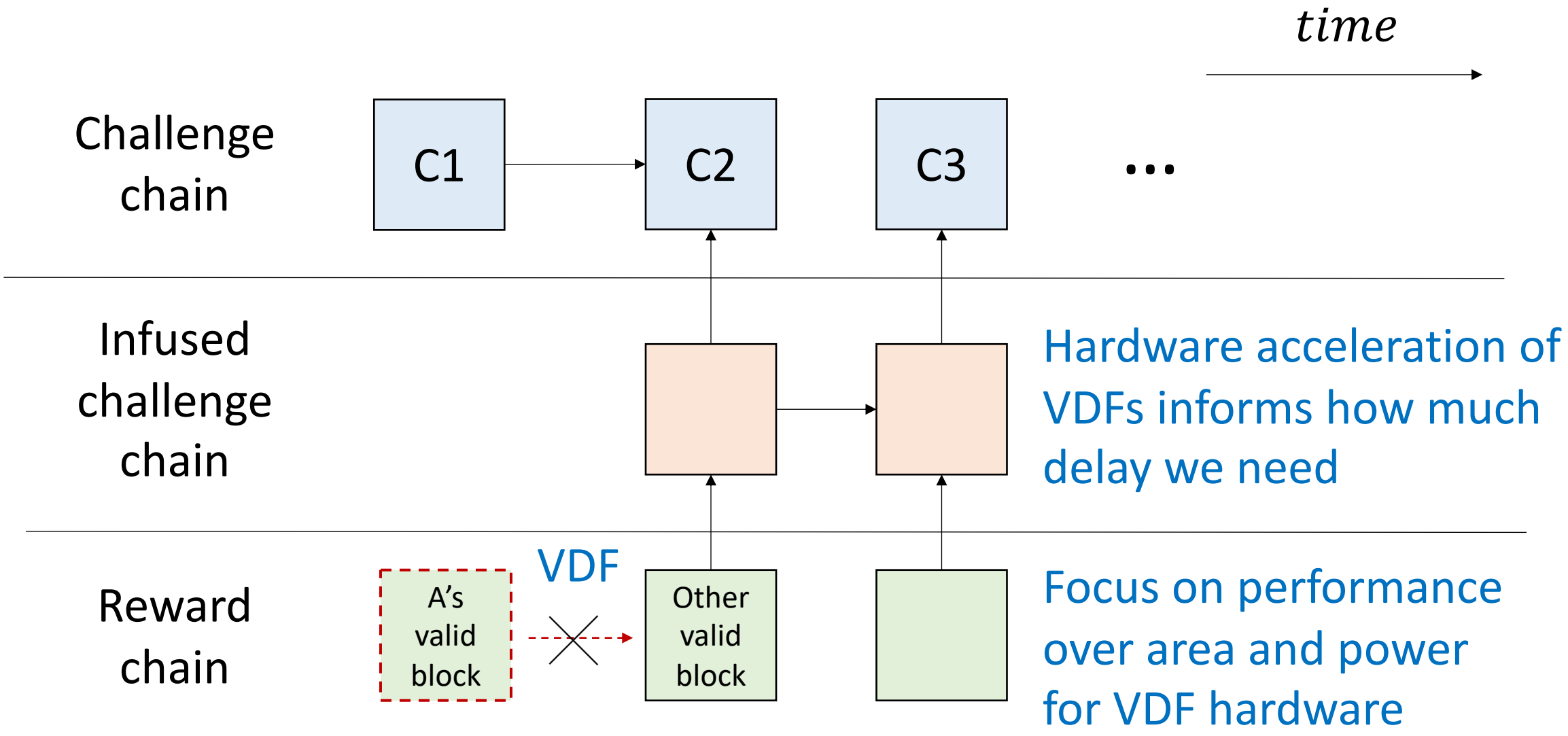
VDFs can secure blockchain systems



VDFs can secure blockchain systems

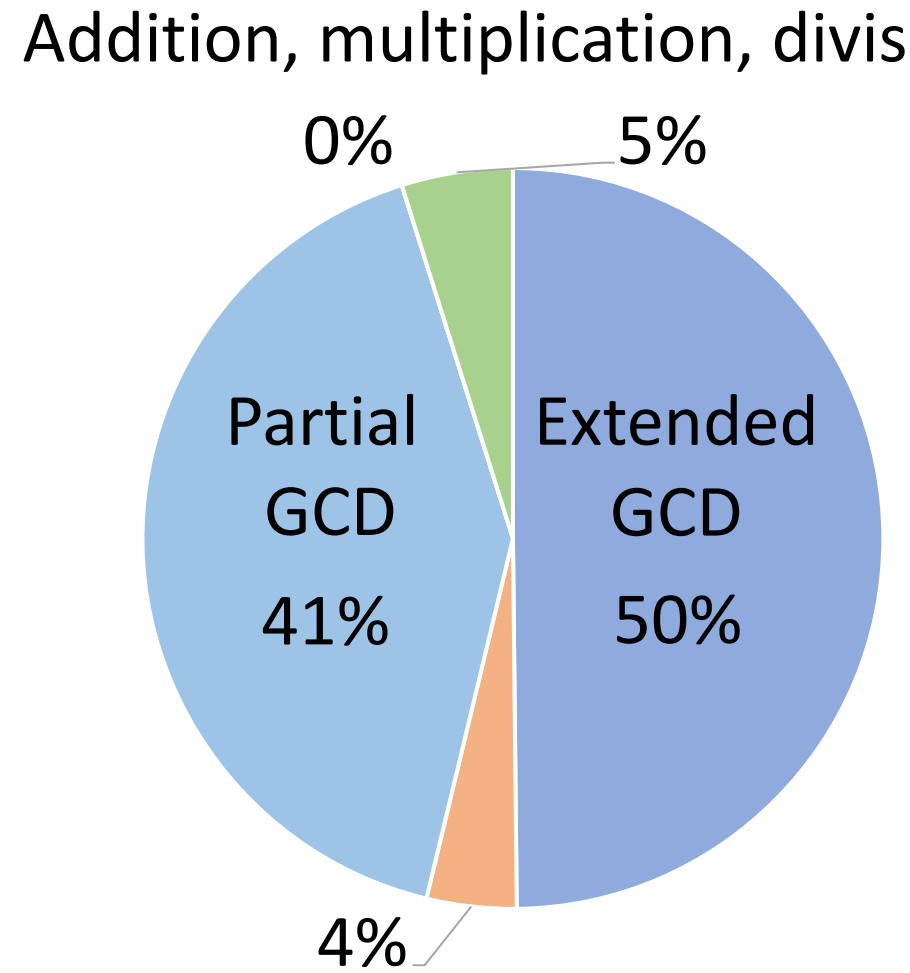


VDFs can secure blockchain systems



VDFs primarily rely on fast extended GCDs

91% of runtime is spent on GCD



GCD Algorithms

Algorithms use GCD-preserving transformations

$$\boldsymbol{g = gcd(a, b) = gcd(a - b, b)}$$

$$a = g * a_g, \qquad b = g * b_g$$

$$\Rightarrow a - b = g * (a_g - b_g)$$

$$3 = gcd(33, 9) = gcd(24, 9)$$

Algorithms use GCD-preserving transformations

$$\boldsymbol{g = gcd(a, b) = gcd(a - b, b)}$$

$$a = g * a_g, \qquad b = g * b_g$$

$$\Rightarrow a - b = g * (a_g - b_g)$$

$$3 = gcd(33, 9) = gcd(24, 9)$$

$$\boldsymbol{gcd(a, b) = gcd(a \bmod b, b)}$$

$$a = g * a_g, \qquad b = g * b_g$$

$$\Rightarrow a \bmod b = a - b * q = g * (a_g - b_g * q)$$

$$3 = gcd(33, 9) = gcd(6, 9)$$

Algorithms use GCD-preserving transformations

Steins $\boldsymbol{g = gcd(a, b) = gcd(a - b, b)}$

$$a = g * a_g, \qquad b = g * b_g$$

$$\Rightarrow a - b = g * (a_g - b_g)$$

$$3 = gcd(33, 9) = gcd(24, 9)$$

Euclid $\boldsymbol{gcd(a, b) = gcd(a \bmod b, b)}$

$$a = g * a_g, \qquad b = g * b_g$$

$$\Rightarrow a \bmod b = a - b * q = g * (a_g - b_g * q)$$

$$3 = gcd(33, 9) = gcd(6, 9)$$

GCD algorithms example: $G(27,2) = 1$

Euclid: $gcd(a \ mod \ b, b)$

a	b	Operation
27	2	original a, b
2	1	27 mod 2 = 1
1	0	2 mod 1 = 0

Euclid's converges faster, but Steins uses simpler operations (shifts, adds, comparisons) instead of division

Steins: shift if even or else $gcd(a - b, b)$

a	b	Operation
27	2	original a, b
27	1	b / 2
26	1	subtract
13	1	a / 2
12	1	subtract
6	1	a / 2
3	1	a / 2
2	1	subtract
1	1	a / 2
1	0	subtract

Extended GCD

- Computes Bezout coefficients b_a, b_b: $b_a * a_0 + b_b * b_0 = \gcd(a_0, b_0)$
- GCD algorithms extended to calculate b_a, b_b by maintaining these relations every cycle where $\gcd(a_0, b_0) = \gcd(a, b)$:

$$u * a_0 + m * b_0 = a$$
$$y * a_0 + n * b_0 = b$$

- At start, $u = n = 1$ and $m = y = 0$
- At end, $\gcd(a_0, b_0) = a$, $b_a = u$, $b_b = m$

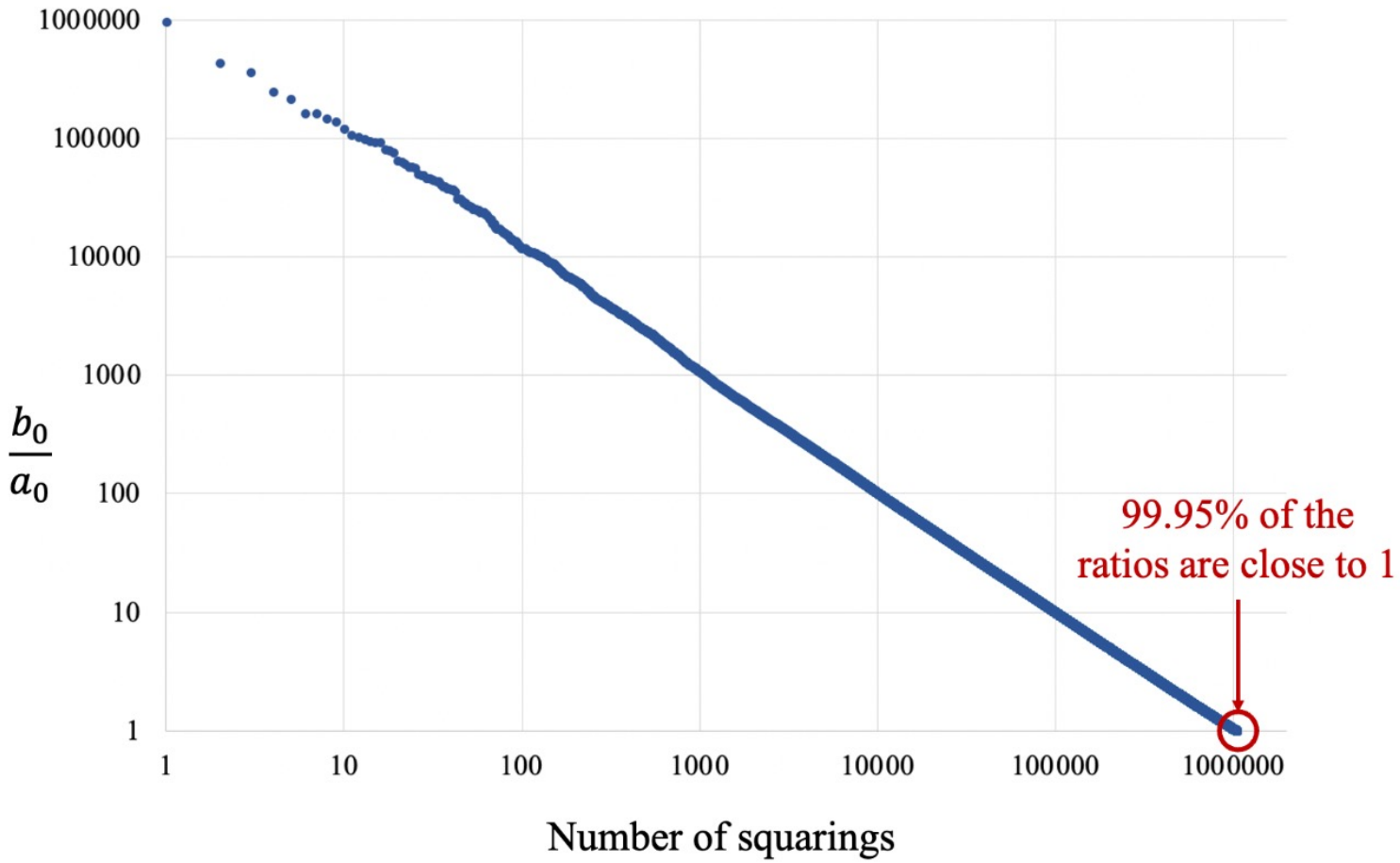
GCD properties in the VDF context

- Bezout coefficients are needed
- Inputs are integers that are
  - Large – 1024+ bits
  - Relatively prime – GCD is known to be equal to 1
- Performance is primary hardware focus

We aim to find state-of-the-art starting points from existing literature that are well-suited to these properties.

VDF inputs are well-suited to Stein's GCD

- Subtraction can reduce many bits when inputs are similar in length



Hardware-friendly GCD algorithms

Avoiding carry propagation for large-integer GCD

- Stein's algorithm has many back-to-back additions by design
- Carry-save adders (CSAs) eliminate carry propagation in these adds
  - $O(1)$ delay instead of $O(number\ of\ bits)$

- Algorithms avoid needing a, b to check $a > b$ by
  - updating $G(a, b) = G\left(\frac{a+b}{2}, \frac{a-b}{2}\right)$ when a, b odd
  - using approximate value of $\log_2 a - \log_2 b$ to decide which branch to take
- Two-bit PM further reduces 2 bits if possible and avoids swapping a, b

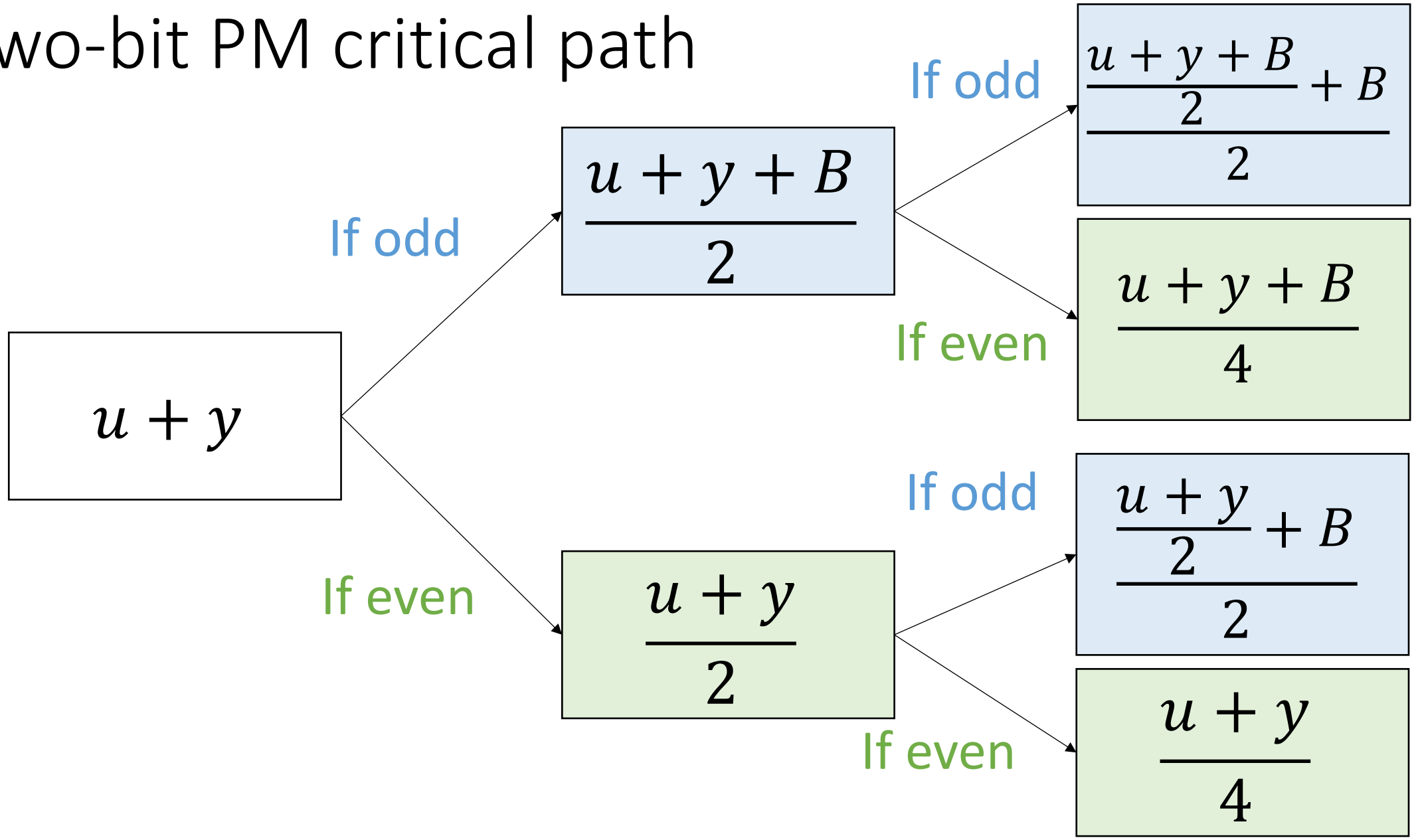
Two-bit Plus-Minus (PM) algorithm

Case Number	Description
1	a is divisible by 4
2	a is divisible by 2 (and not by 4)
3	b is divisible by 4
4	b is divisible by 2 (and not by 4)
5	$\delta \geq 0$ and $a + b$ is divisible by 4
6	$\delta \geq 0$ and $b - a$ is divisible by 4
7	$\delta < 0$ and $a + b$ is divisible by 4
8	$\delta < 0$ and $b - a$ is divisible by 4

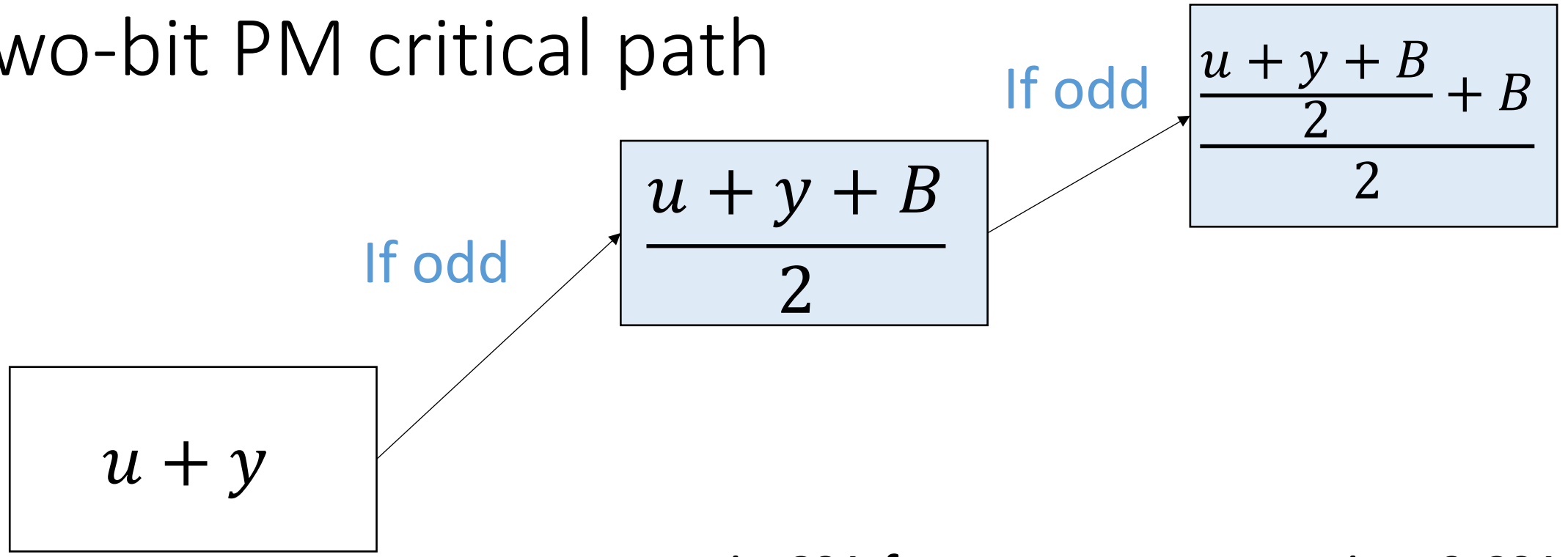
Bits reduced per cycle

- Worst case: 1
- Best case: $2 + \min(\log_2 a\,, \log_2 b)$
- Reduces more bits than Stein's 80% of the time

Two-bit PM critical path

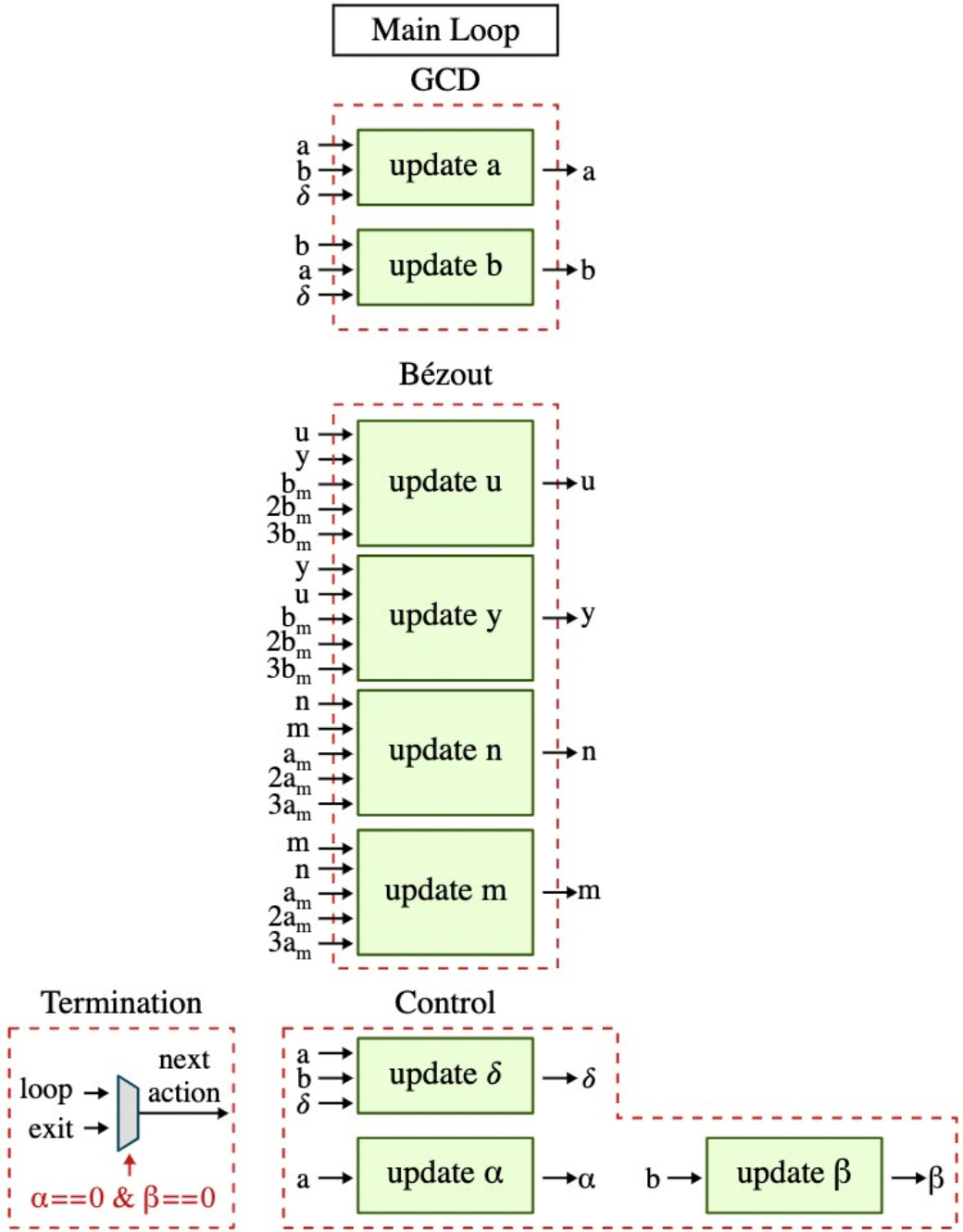


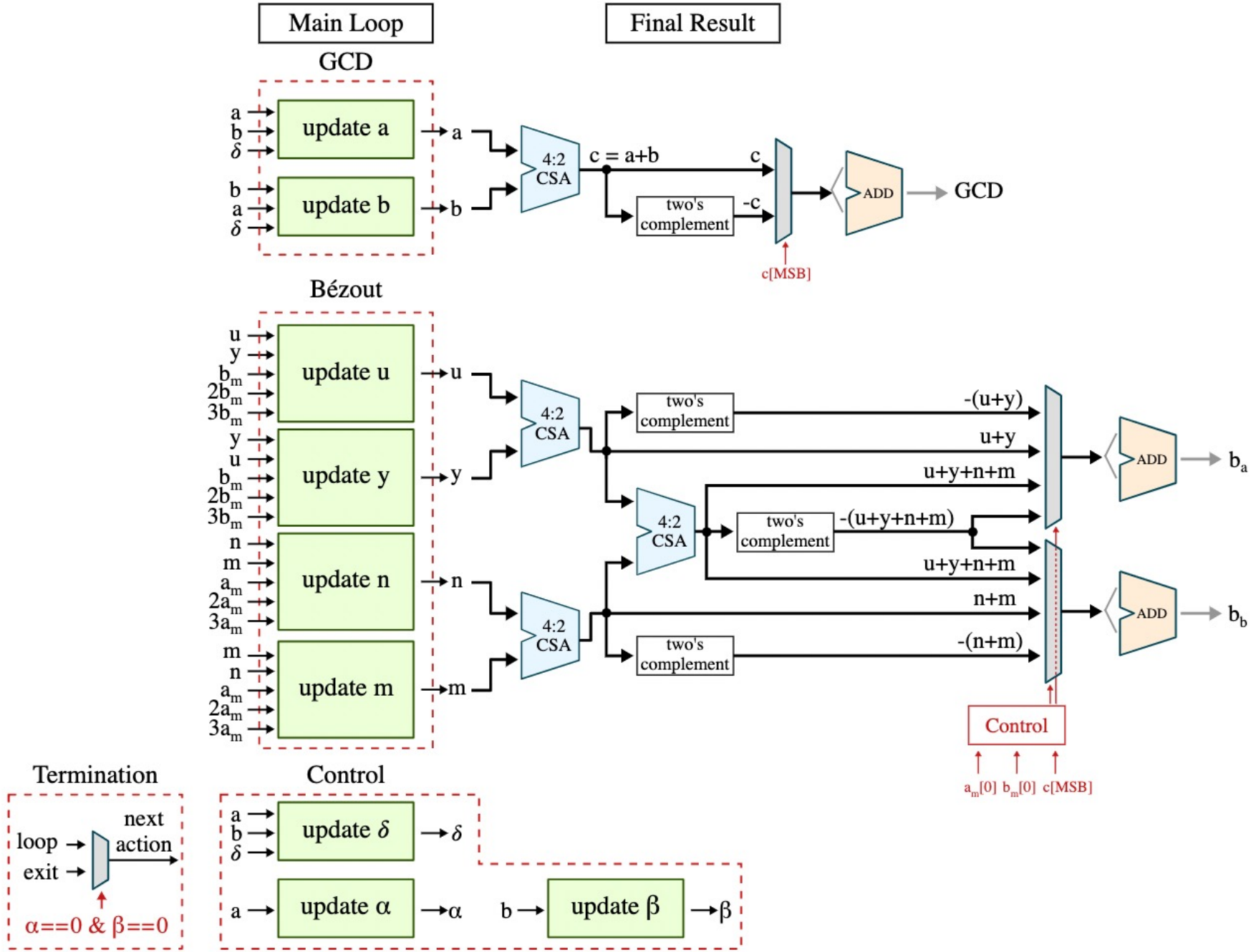
Two-bit PM critical path

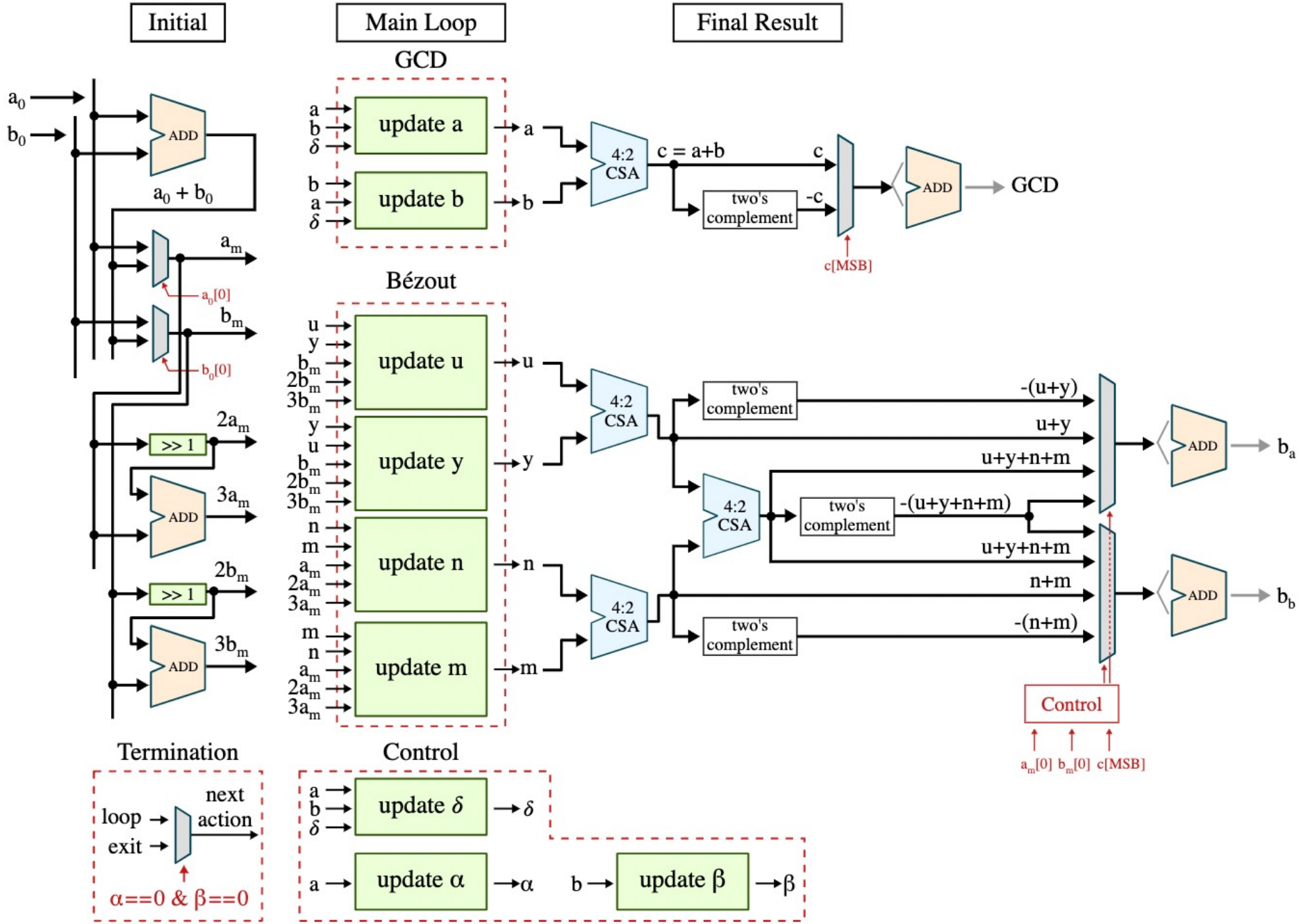


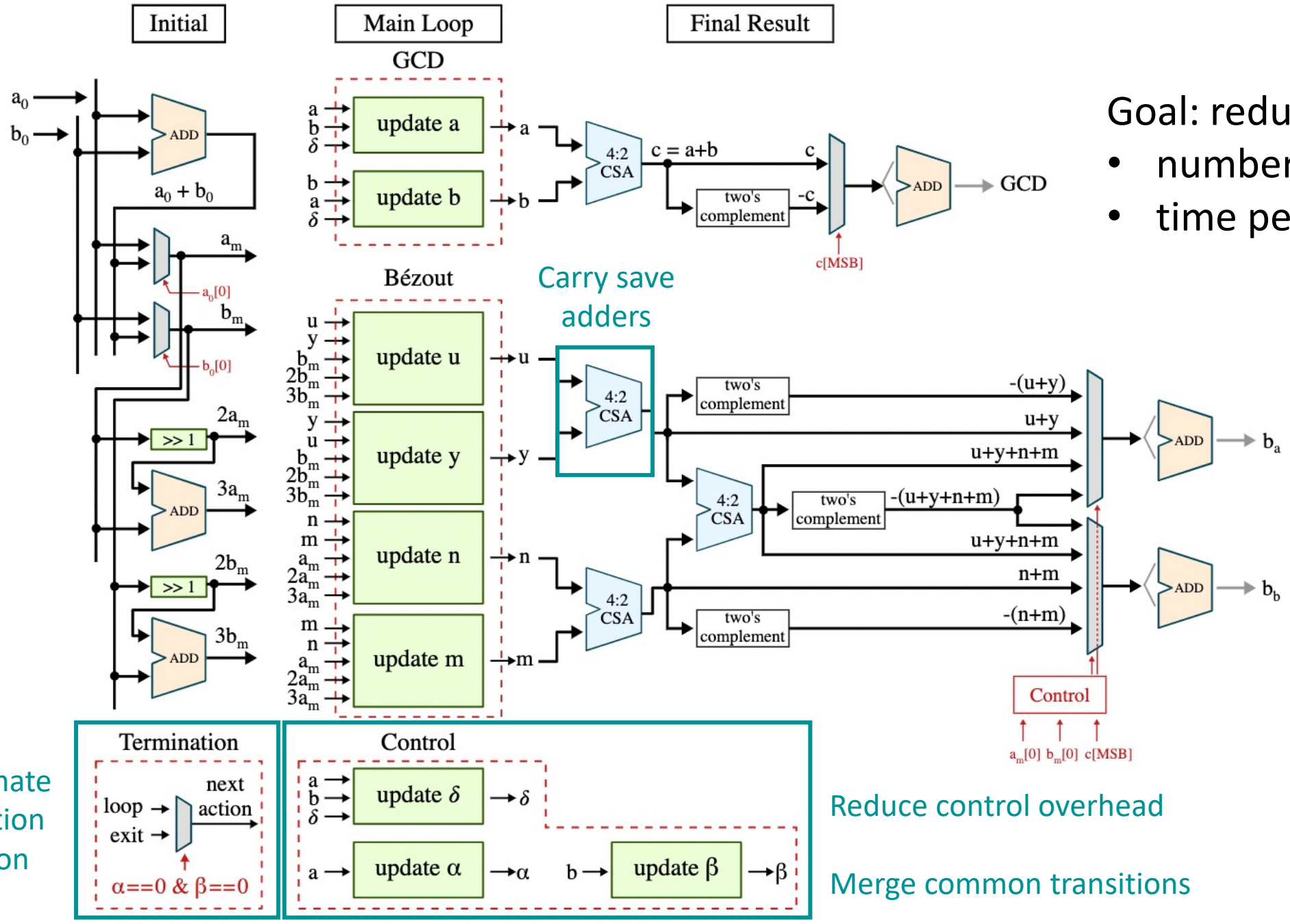
- u, y are in CSA form, so $u + y$ requires 2 CSAs
- B is not in CSA form, so critical path has **4 CSAs**

Our GCD Design


Main Loop
GCD
a
b
δ
update a
a
b
a
δ
update b
b
Bézout
u
y
b_m
$2b_m$
$3b_m$
update u
u
y
u
b_m
$2b_m$
$3b_m$
update y
y
n
m
a_m
$2a_m$
$3a_m$
update n
n
m
n
a_m
$2a_m$
$3a_m$
update m
m
Termination
loop
exit
next
action
α==0 & β==0
Control
a
b
δ
update δ
δ
a
update α
α
b
update β
β


Main Loop
Final Result
GCD
update a
update b
4:2 CSA
c = a+b
two's complement
-c
c
c[MSB]
ADD
GCD
Bézout
update u
update y
update n
update m
-(u+y)
u+y
u+y+n+m
-(u+y+n+m)
n+m
-(n+m)
Control
a_m[0] b_m[0] c[MSB]
b_a
b_b
Termination
loop
exit
next action
α==0 & β==0
update δ
update α
update β


Initial
Main Loop
Final Result
GCD
Bézout
Termination
Control
ADD
>> 1
update a
update b
update u
update y
update n
update m
update δ
update α
update β
4:2 CSA
two's complement
c = a+b
-c
c[MSB]
GCD
-(u+y)
u+y
u+y+n+m
-(u+y+n+m)
n+m
-(n+m)
Control
loop
exit
next action
α==0 & β==0



Goal: reduce

- number of cycles
- time per cycle

Carry save adders

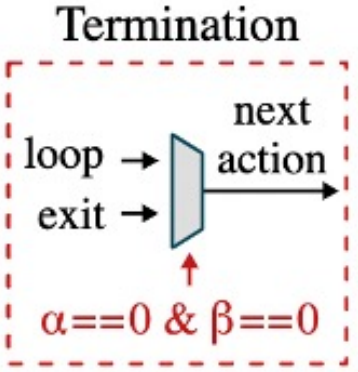
Approximate termination condition

Reduce control overhead

Merge common transitions

Terminate based on approximate binary logs

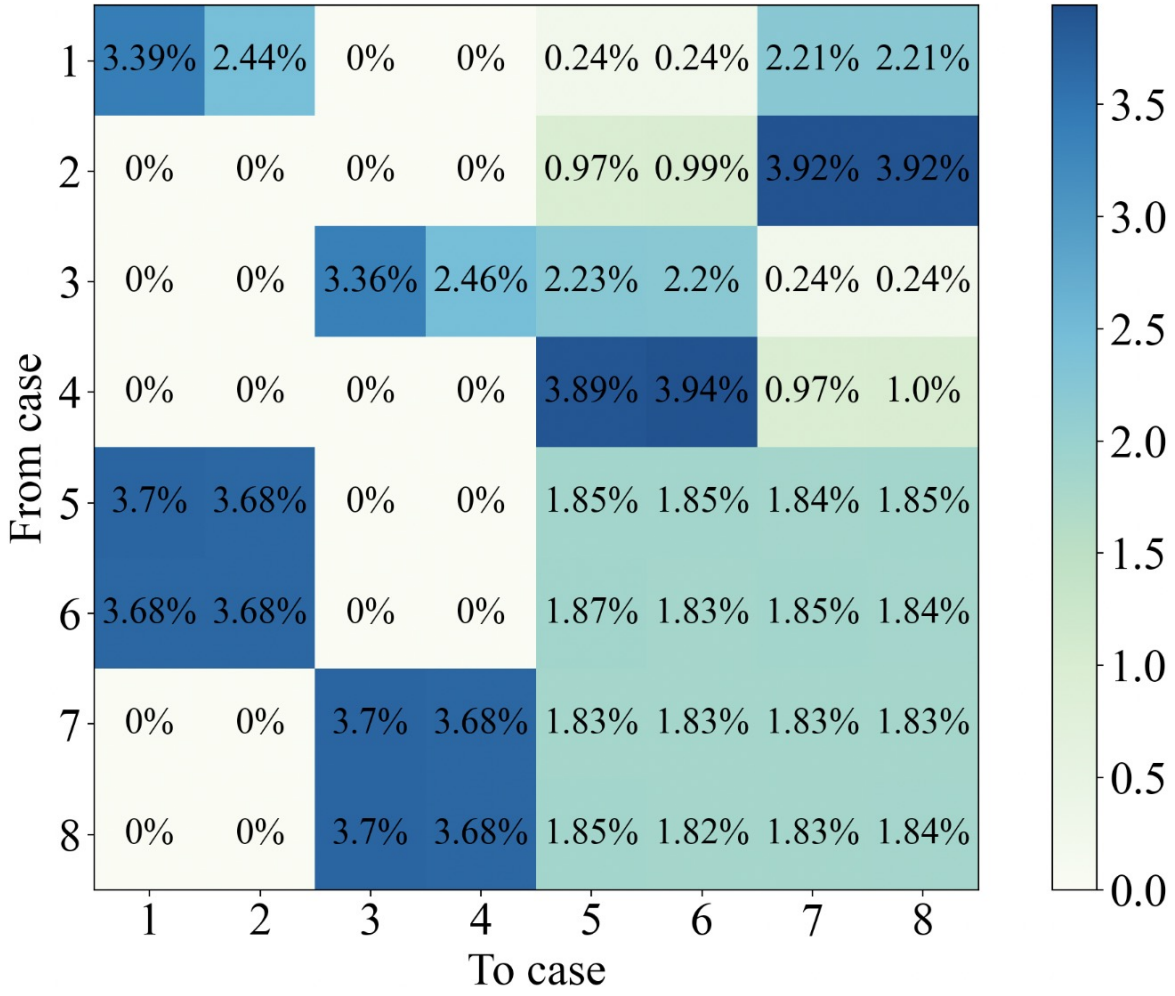
- We check if $\alpha \cong \log_2 a, \beta \cong \log_2 b$ are 0 instead of a, b to
  - Avoid the carry-propagate add to get a, b from CSA form



- However, this is an approximation and adds 154 cycles
- So, we get a and b occasionally with carry-propagate adds to compute $\alpha = \log_2 a, \beta = \log_2 b$
- This occasional correction reduces the overhead to 43 cycles (3.5%)

Performance impact: Reduce 2 long 1024-bit carry-propagate checks to 2 faster 10-bit carry-propagate checks

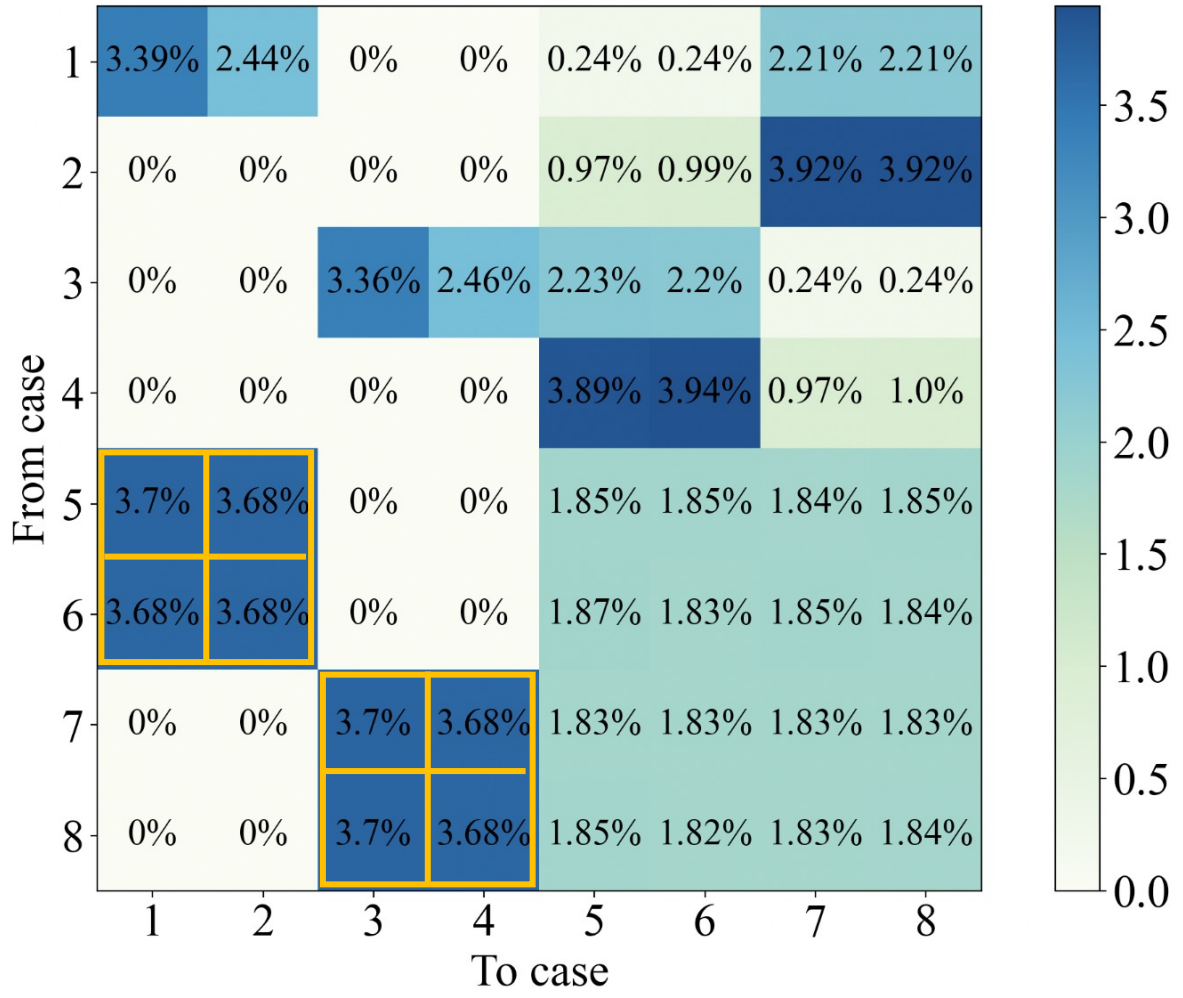
Reducing cycle count by merging transitions



Two-bit PM Transition Matrix

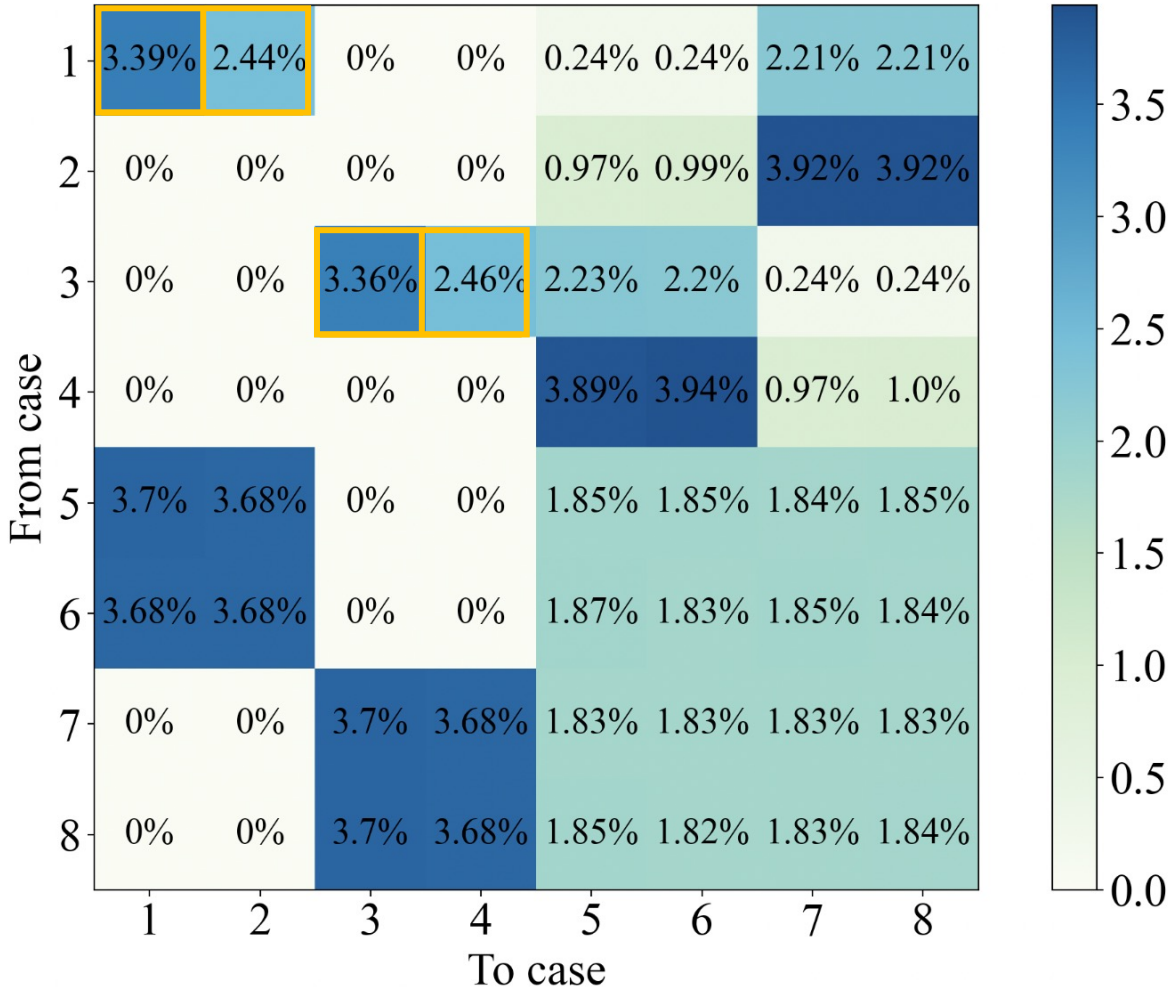
	Description
1	a is divisible by 4
2	a is divisible by 2 (and not by 4)
3	b is divisible by 4
4	b is divisible by 2 (and not by 4)
5	$\delta \geq 0$ and $a + b$ is divisible by 4
6	$\delta \geq 0$ and $b - a$ is divisible by 4
7	$\delta < 0$ and $a + b$ is divisible by 4
8	$\delta < 0$ and $b - a$ is divisible by 4

Merging common transitions



	Description
1	a is divisible by 4
2	a is divisible by 2 (and not by 4)
3	b is divisible by 4
4	b is divisible by 2 (and not by 4)
5	$\delta \geq 0$ and $a + b$ is divisible by 4
6	$\delta \geq 0$ and $b - a$ is divisible by 4
7	$\delta < 0$ and $a + b$ is divisible by 4
8	$\delta < 0$ and $b - a$ is divisible by 4

Include divide by 8, 16 cases



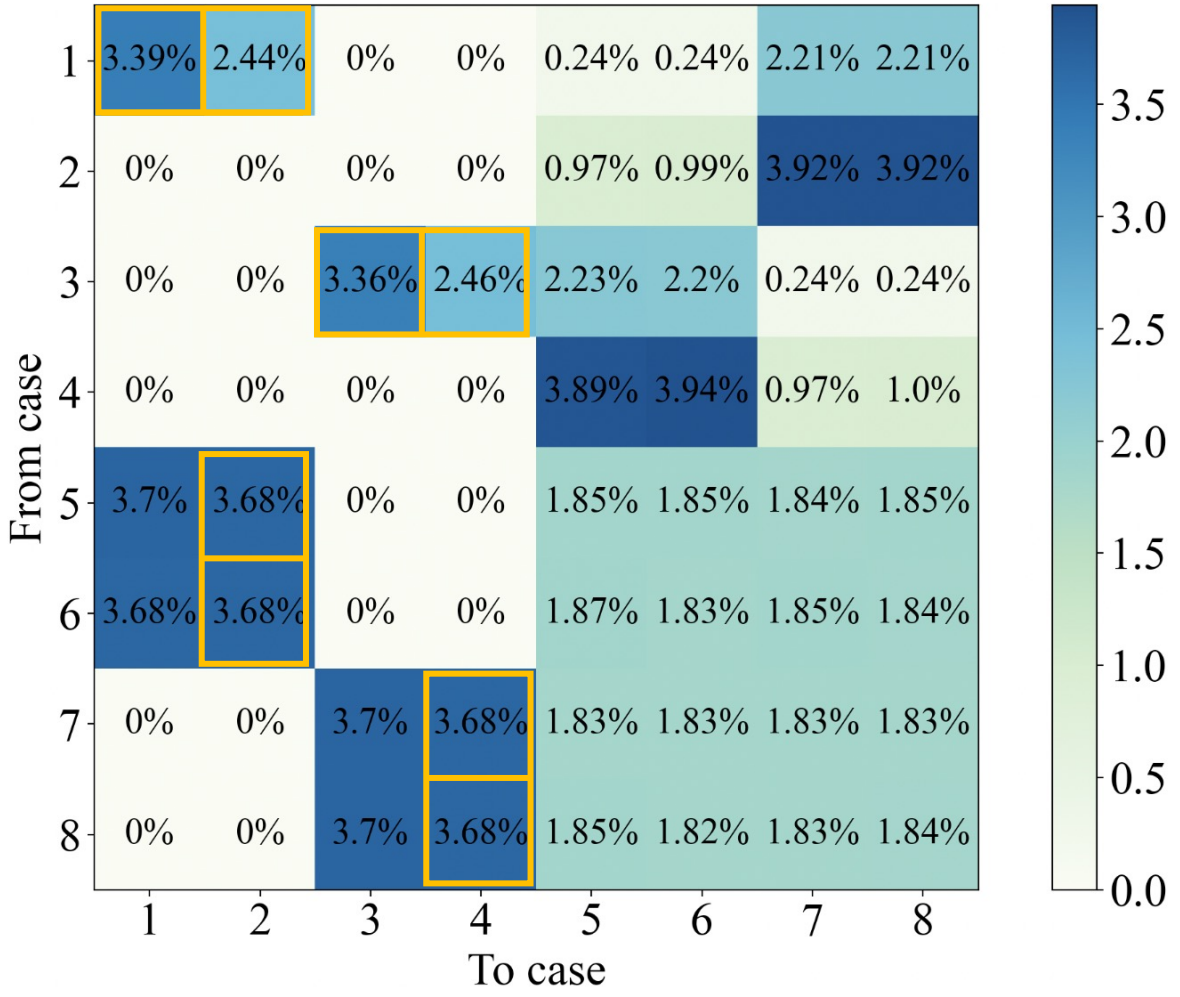
	Description
1	a is divisible by 4
2	a is divisible by 2 (and not by 4)
3	b is divisible by 4
4	b is divisible by 2 (and not by 4)
5	$\delta \geq 0$ and $a + b$ is divisible by 4
6	$\delta \geq 0$ and $b - a$ is divisible by 4
7	$\delta < 0$ and $a + b$ is divisible by 4
8	$\delta < 0$ and $b - a$ is divisible by 4

Reducing cycle count summary

Transition Optimization	Cycles	Datapath (ns)	GCD Runtime (us)
Baseline	0	0	3.70
Merge 5-2 into 5 **Merge 7-4 into 7**	**-236**	**+0.6 (1 CSA)**	**3.60**
Merge 5-2, 5-1 into 5 Merge 7-4, 7-3 into 7	-374	+1.2 (2 CSAs)	3.62
Merge 5-2, 5-1, 5-1-1, 5-1-2 Merge 7-4, 7-3, 7-3-3, 7-3-4	-464	+2.4 (4 CSAs)	4.17
Add cases: a, b divisible by 8	-65	0	3.52
Add cases: a, b divisible by 16, 8	**-92**	**0**	**3.44**
Bolded rows	-328	+0.6 (1 CSA)	3.27

Performance impact: 11.6% faster runtime by trading a small increase in cycle time for lower cycle counts

Reducing cycle count chosen mergings



	Description
1	a is divisible by 4
2	a is divisible by 2 (and not by 4)
3	b is divisible by 4
4	b is divisible by 2 (and not by 4)
5	$\delta \geq 0$ and $a + b$ is divisible by 4
6	$\delta \geq 0$ and $b - a$ is divisible by 4
7	$\delta < 0$ and $a + b$ is divisible by 4
8	$\delta < 0$ and $b - a$ is divisible by 4

Other optimizations

CSA-related

- Repeatedly shifting
- Preserving sign
- Reducing number of CSAs
- Slower clock for carry-propagates

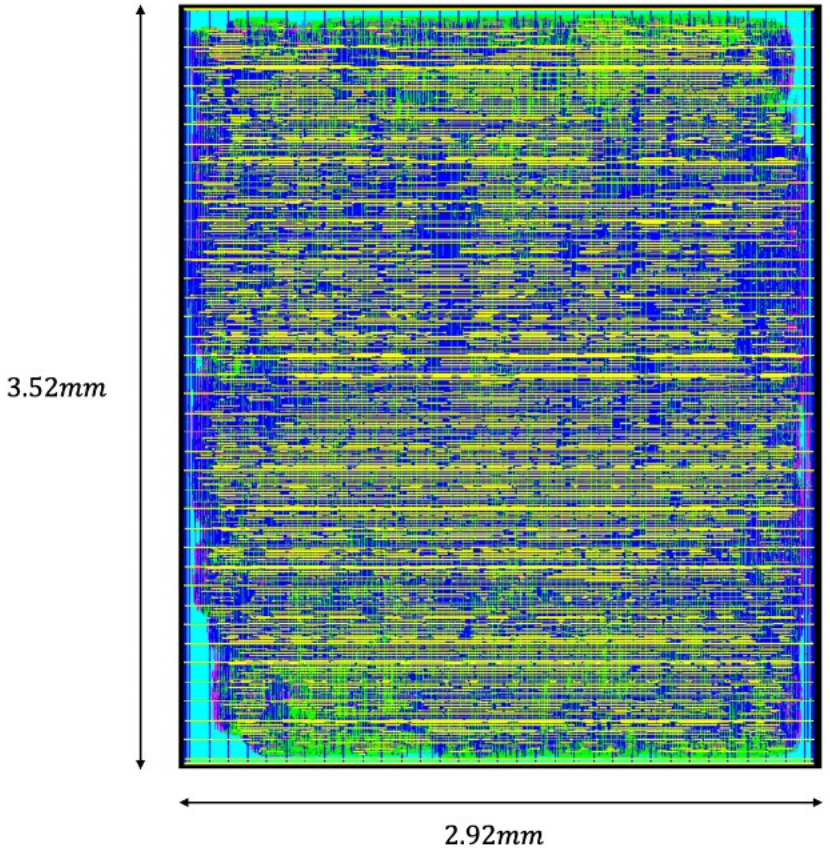
Control overhead

- Precomputing control signals with high fanout
- Late select

Performance Impact:

- **Number of carry-propagate adds is O(1) instead of O(number of iterations)**
- **Eliminate buffer delay to drive high fanout signals from critical path**

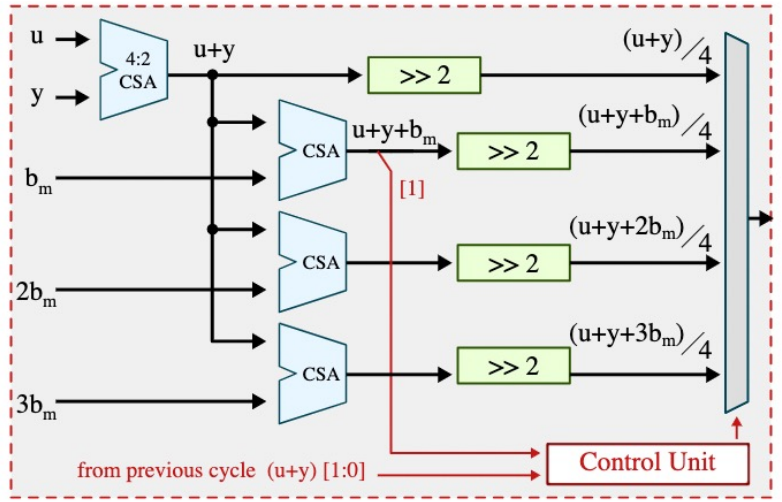
SKY130 prototype with Efabless



Technology	SKY 130nm
Library	High density
Voltage	1.8 V
Frequency	325.4 MHz
Synthesized Area	5.66 mm^2
Chip Area	10.28 mm^2
Density	57%

Critical Path: 3ns = 35 FO4 Inv Delay

Operation	Standard Cell	Delay (ns)	FO4 Inv Delay
DFF clk to Q	dfrtp_2	0.54	6.28
4:2 CSAs for $u + y$	fa_2	0.51	5.93
	fa_2	0.67	7.79
CSA for $(u + y) + 3 * b_m$	fa_2	0.45	5.23
	clkbuf_1	0.11	1.28
Half adder to correct +1	and2_4	0.13	1.51
	nor2_1	0.09	1.05
4:1 mux	mux2i_1	0.07	0.81
	a211oi_1	0.23	2.67
8:1 mux	nand2_1	0.07	0.81
	nand3_1	0.07	0.81
Library setup time	dfrtp_1	0.07	0.81
Total	N/A	3	35



Area Breakdown

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: δ, α, β update	0.57	10%
Final result calculation	0.36	6.4%
JTAG	0.22	3.6%
Miscellaneous	0.10	1.7%
Total	5.66	100%

Speedup achieved over prior work

Implementation	**Technology**	**Algorithm**	**VDF Squaring Speedup***	**Extended GCD Speedup**
Zhu et al.'s VDF accelerator	28 nm	Chia's first competition round reference	10X	14X
Chia Network's C++ based on competition first round results	5 nm (M1)	NUDUPL	9X	67X

*assuming only GCD is accelerated in hardware

We can design efficient GCD accelerators for VDFs

- Verifiable delay functions are an exciting new cryptographic primitive with applications in blockchain
- Extended GCD requires 91% of the total VDF runtime
- Our design speeds up Chia Network's C++ by 9X

VDFs can be sped up by an order of magnitude with our GCD hardware design techniques, laying the groundwork for fast hardware-accelerated VDFs in future blockchain systems.