A Fast Large-Integer Extended GCD Algorithm and Hardware Design for Verifiable Delay Functions and Modular Inversion

Kavya Sreedhar, Mark Horowitz, Christopher Torng

skavya@stanford.edu

AHA Affiliates Meeting

May 4th, 2022

Cryptography relies on hard problems

- Modern cryptography is based on computationally hard problems
  - Typically requiring large-integer arithmetic
- Execution time of problems is critical
  - Re-evaluate with algorithmic and hardware advances
- Recent application developments motivate revisiting XGCD

Verifiable delay functions (VDFs) [1]

- VDFs require slow evaluation but fast verification
  - Require fixed amount of sequential work to be evaluated
  - Output a unique result that is still efficiently verifiable

- Computationally hard problem can be a trapdoor function
  - $y = f(x)$ is easy to compute
  - $x = g(y)$ is difficult to compute without some secret s and $f(s)$

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

Verifiable delay functions (VDFs) [1]

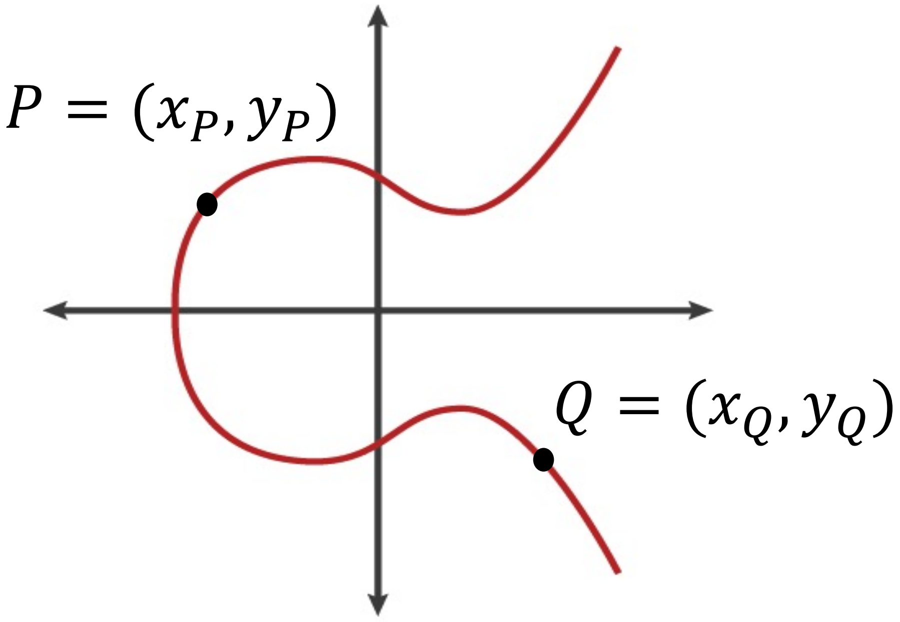
- VDFs are increasingly being used in blockchain systems
- The VDF adopted by Chia spends 90+% of execution time on XGCDs
  - Inputs are large (1024+ bits) and not secret

Verifiable delays are useful to secure blockchain systems, and their performance determines VDF security levels.

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

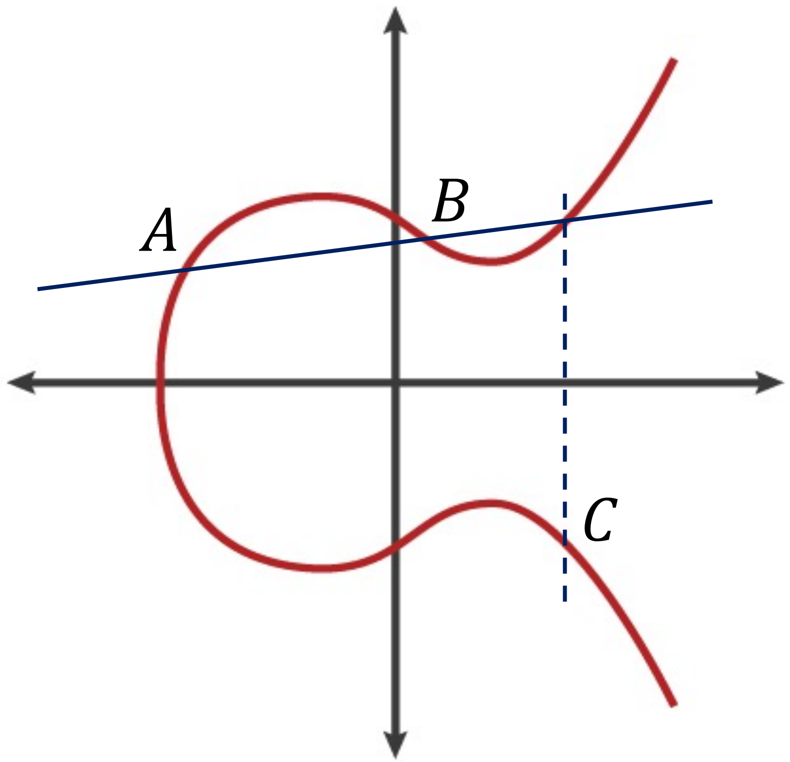
Elliptic Curve Cryptography (ECC)

- Used for public key authentication
- Construction has points (x, y) : $By^2 = x^3 + Ax^2 + x$
  - A, B, x, y can be integers mod p



Elliptic Curve Cryptography (ECC)

- Computationally hard problem
  - Given P, Q on the curve, find $k \in Z$ such that $[k]P = Q$
    - Points on curve (x, y) are integers mod p



Elliptic Curve Cryptography (ECC)

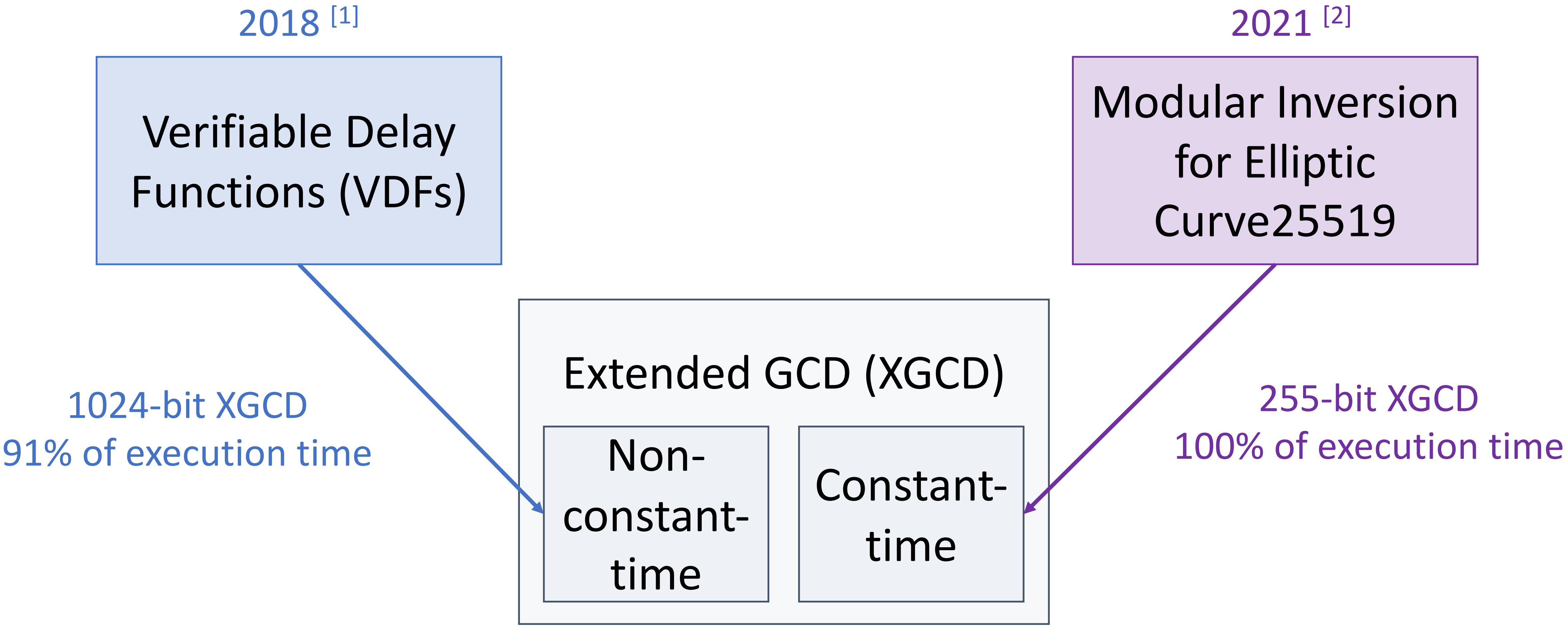
- Most time-consuming operation is modular inversion
  - Find x^{-1} such that $x * x^{-1} = 1\ (mod\ p)$
  - Since x is secret, this must be constant-time

- Recently, XGCD was found to be the fastest way to do this [2]

ECC arithmetic now relies on XGCD, motivating a need for faster XGCD and reconsidering algorithms with many inversions.

[2] Bernstein and Yang. Fast constant-time gcd computation and modular inversion. CHES 2019.

Application Summary



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

How fast can one do XGCD?

- GCD is a fundamental operation in number theory and cryptography
  - Many algorithms developed in the 1980s/90s
  - More recently, software GCD libraries have been highly tuned

- However, few works have implemented extended GCD in hardware

Can we significantly improve XGCD performance with hardware?

XGCD accelerator design space

- **Optimal algorithmic choice for hardware**
- Large-integer arithmetic circuit optimizations
- Different application requirements

Prior hardware work:
Builds from
division-based
algorithms

Our ASIC design:
Builds from
subtraction-based
algorithms

XGCD accelerator design space

- Optimal algorithmic choice for hardware
- **Large-integer arithmetic circuit optimizations**
- Different application requirements

Prior hardware work:
Directly adds large integers or suggests using carry-save adders

Our ASIC design:
Uses carry-save adders and addresses related challenges

XGCD accelerator design space

- Optimal algorithmic choice for hardware
- Large-integer arithmetic circuit optimizations
- **Different application requirements**

Prior hardware work: provides point solutions targeting an application space

Our ASIC design: Can evaluate fast average and constant-time XGCD

Algorithms use GCD-preserving transformations

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

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

Algorithms use GCD-preserving transformations

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

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

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

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

Algorithms use GCD-preserving transformations

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

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

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

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

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 $GCD(27,2) = 1$

Euclid

a	b	Operation
27	2	start
2	1	27 mod 2
1	0	2 mod 1

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

Euclid

a	b	Operation
27	2	start
2	1	27 mod 2
1	0	2 mod 1

Stein [1]

a	b	Operation
27	2	start
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

[1] Josef Stein. Computational problems associated with Racah Algebra. Journal of Computational Physics 1967

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

Euclid

a	b	Operation
27	2	start
2	1	27 mod 2
1	0	2 mod 1

Stein [1]

a	b	Operation
27	2	start
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

Two-bit Plus-Minus (PM) [2]

a	b	Operation
27	2	original a, b
27	1	b / 2
7	1	(a + b) / 4
2	1	(a + b) / 4
1	1	a / 2
1	0	(a – b) / 4

[1] Josef Stein. Computational problems associated with Racah Algebra. Journal of Computational Physics 1967.
[2] Yun and Zhang. A fast carry-free algorithm and hardware design for extended integer gcd computation. ACM Symposium on Symbolic and Algebraic Computation 1986.

Extended GCD (XGCD)

- Computes Bézout coefficients satisfying Bézout Identity

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

- Maintains these relations each cycle, where $\gcd(a_0, b_0) = \gcd(a, b)$

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

Which approach is better in hardware?

- Goal: minimize execution time = iteration time $*$ number of iterations

Which approach is better in hardware?

- Goal: minimize execution time = iteration time $*$ number of iterations

  cycle time $*$ number of cycles

- Does the answer change for fast average vs constant-time execution?

Comparing number of iterations

- Worst-case number of iterations for 255-bit inputs
  - Euclid 283
  - Two-bit PM 284

1X

Two-bit PM will be faster

Comparing number of iterations

- Worst-case number of iterations for 255-bit inputs
  - Euclid 283
  - Two-bit PM 284

1X

Two-bit PM will be faster

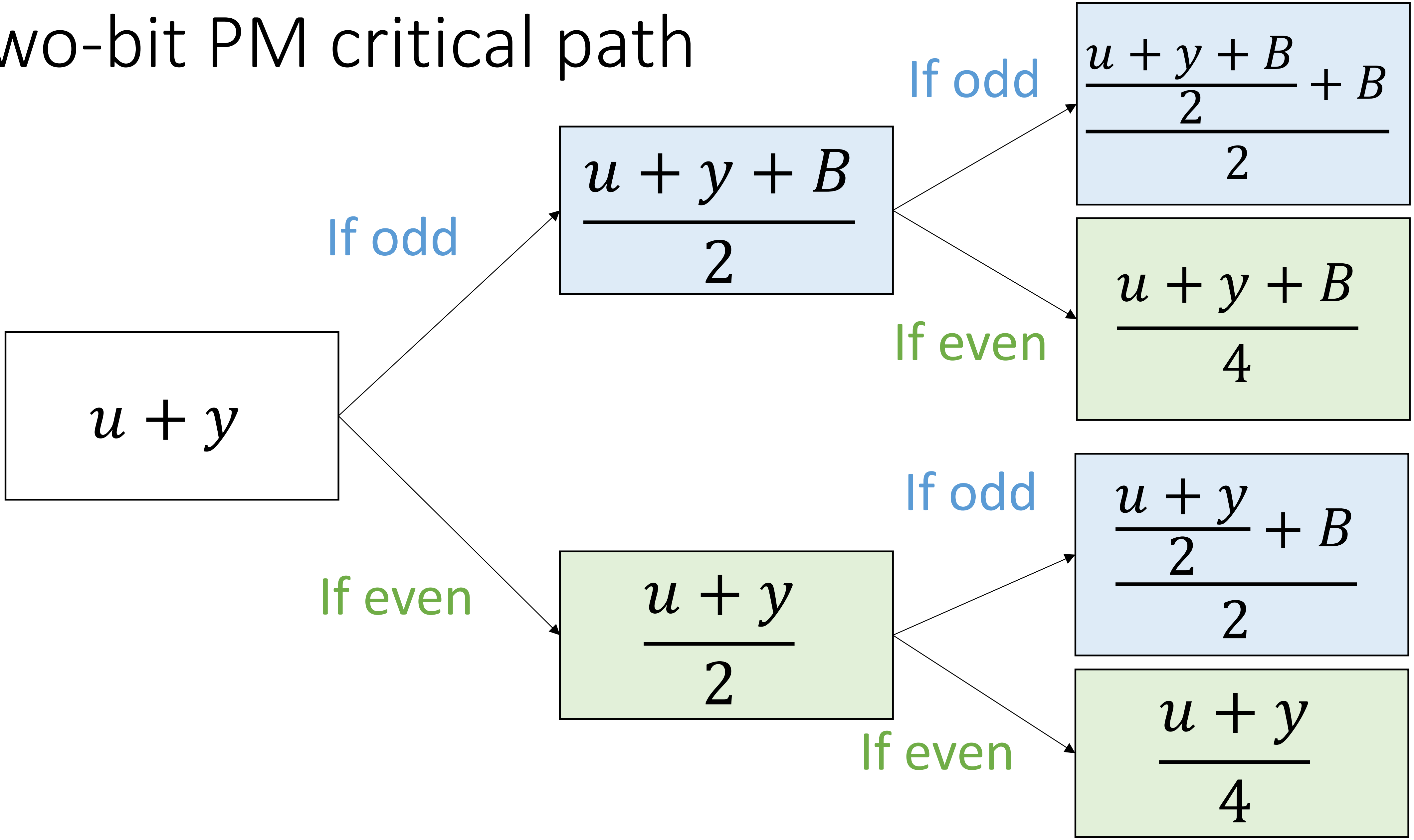
- Average number of iterations for 1024-bit inputs
  - Euclid 598
  - Stein 2163
  - Two-bit PM 1195

3.6X

2X

Can two-bit PM critical path be 2X shorter than Euclid’s?

Two-bit PM critical path



Two-bit PM critical path

$$u + y$$

If odd

$$\frac{u + y + B}{2}$$

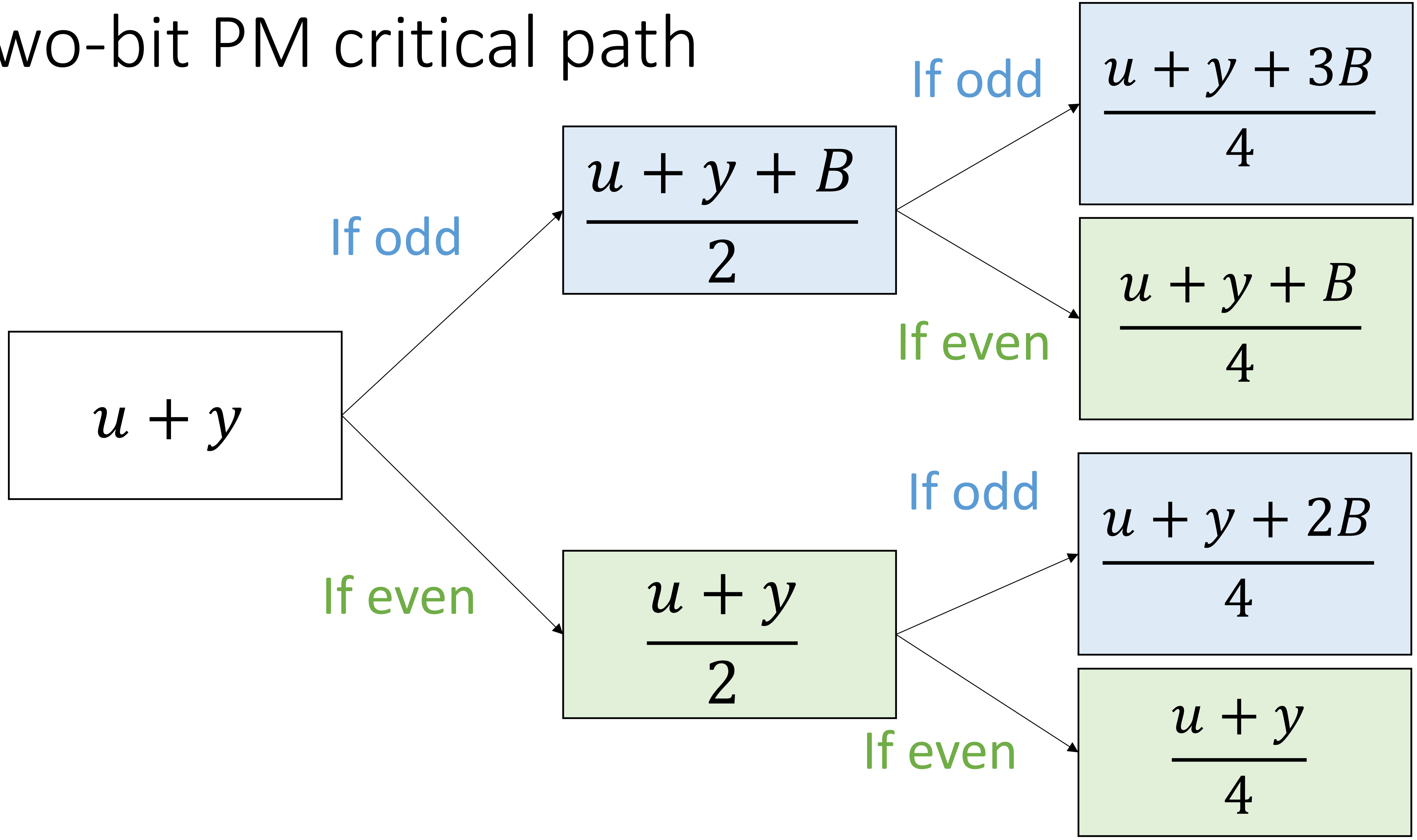
If odd

$$\frac{\frac{u + y + B}{2} + B}{2}$$

Can be rewritten as

$$\frac{u + y + 3B}{4}$$

Two-bit PM critical path

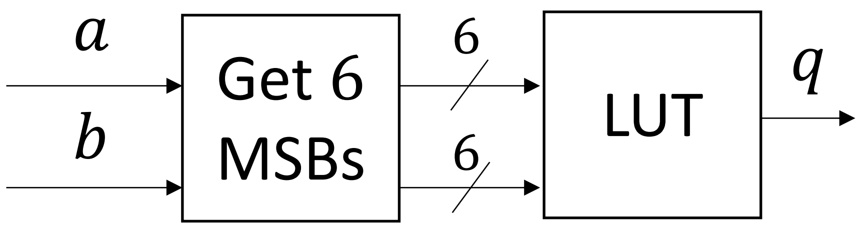


Euclid critical path

Compute $q \leq \lfloor \frac{a}{b} \rfloor$ $\longrightarrow$ Compute $q * b$ $\longrightarrow$ Compute a $- q * b$

Euclid critical path

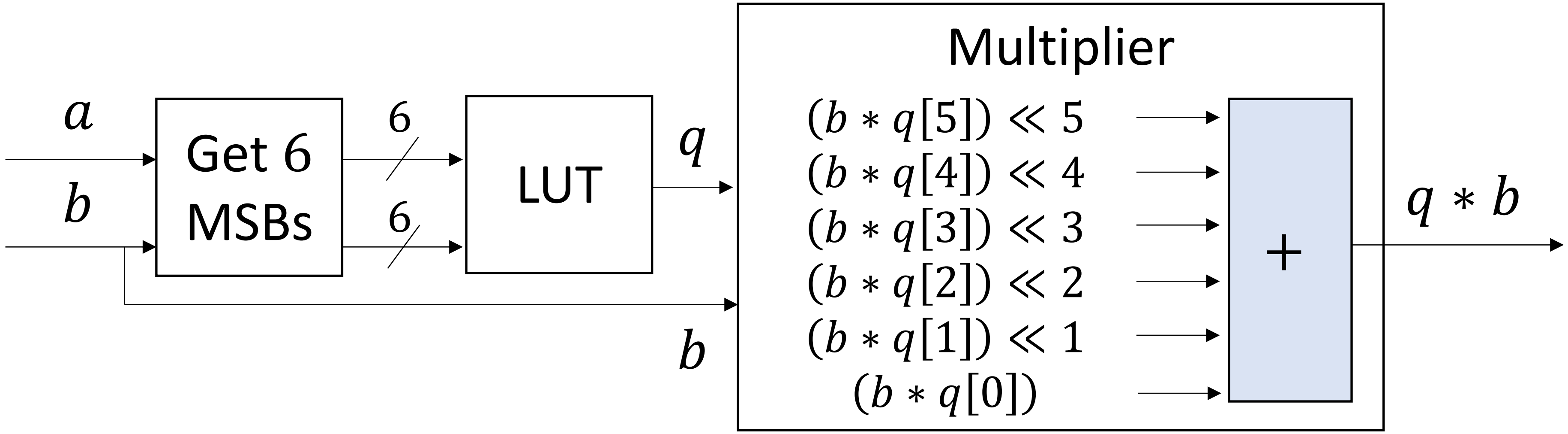
Compute $q \leq \lfloor \frac{a}{b} \rfloor$ ⟶ Compute $q * b$ ⟶ Compute a $- q * b$



- Most quotients in Euclid's algorithm are small for 1024-bit inputs
- Can estimate few of the most significant bits of q for faster execution

Euclid critical path

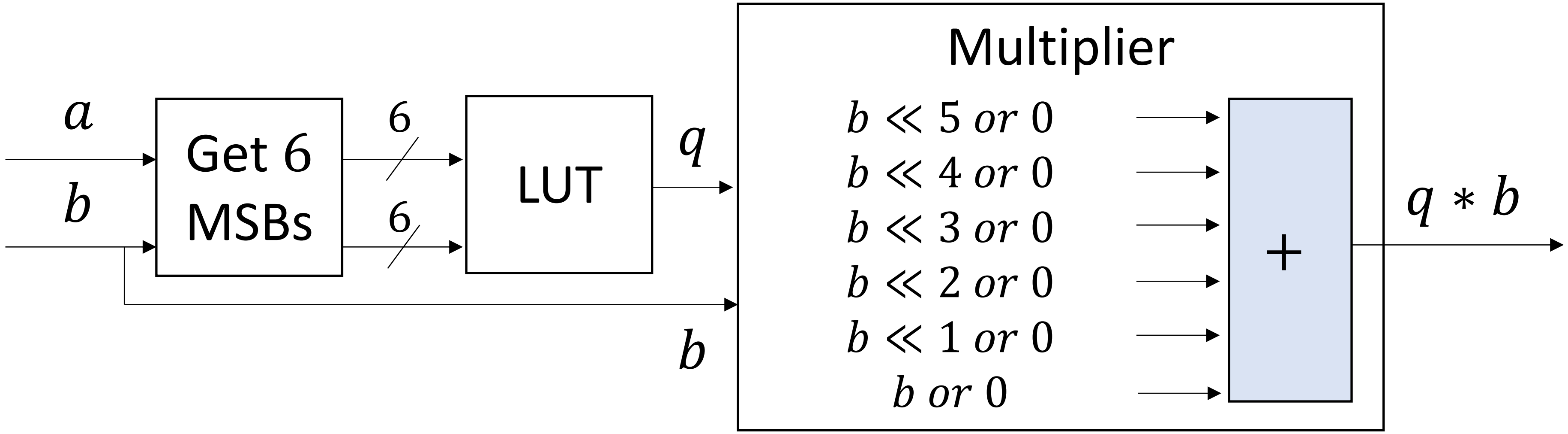
Compute $q \leq \lfloor \frac{a}{b} \rfloor$ $\longrightarrow$ Compute $q * b$ $\longrightarrow$ Compute a $- q * b$



- Most quotients in Euclid’s algorithm are small for 1024-bit inputs
- Can estimate few of the most significant bits of q for faster execution

Euclid critical path

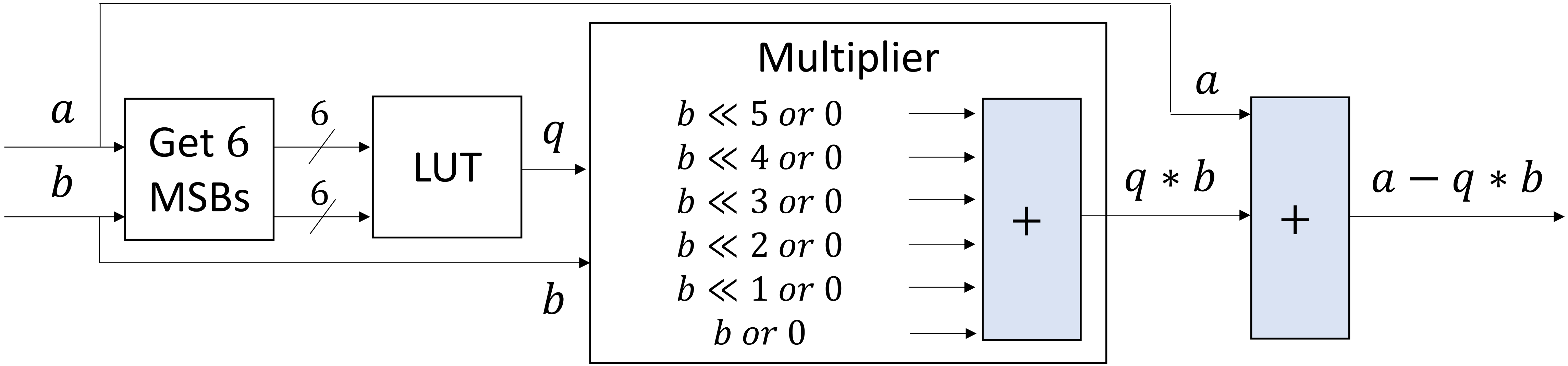
Compute $q \leq \lfloor \frac{a}{b} \rfloor$ ⟶ Compute $q * b$ ⟶ Compute a $- q * b$



- Most quotients in Euclid’s algorithm are small for 1024-bit inputs
- Can estimate few of the most significant bits of q for faster execution

Euclid critical path

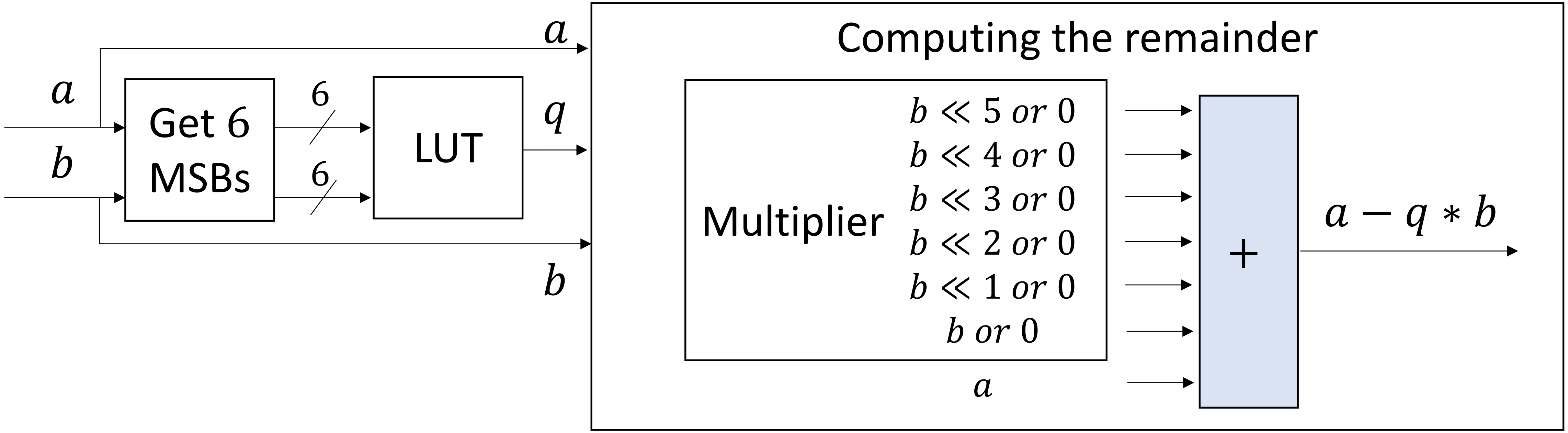
Compute $q \leq \lfloor \frac{a}{b} \rfloor$ ⟶ Compute $q * b$ ⟶ Compute a $- q * b$



- Most quotients in Euclid's algorithm are small for 1024-bit inputs
- Can estimate few of the most significant bits of q for faster execution

Euclid critical path

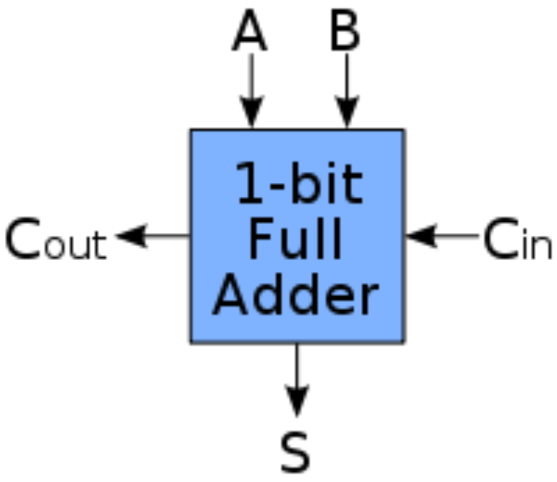
Compute $q \leq \lfloor\frac{a}{b}\rfloor$ ⟶ Compute $q * b$ ⟶ Compute $a - q * b$



Critical paths primarily require additions

- 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

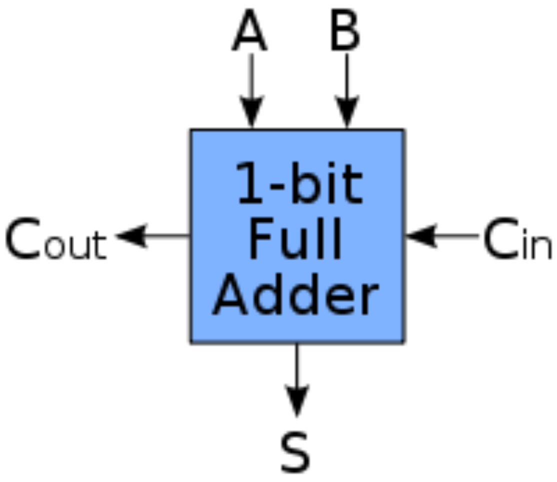
```
    1101 (a)
    1111 (b)
+   0010 (c)
-------------
    0000
```



Critical paths primarily require additions

- 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

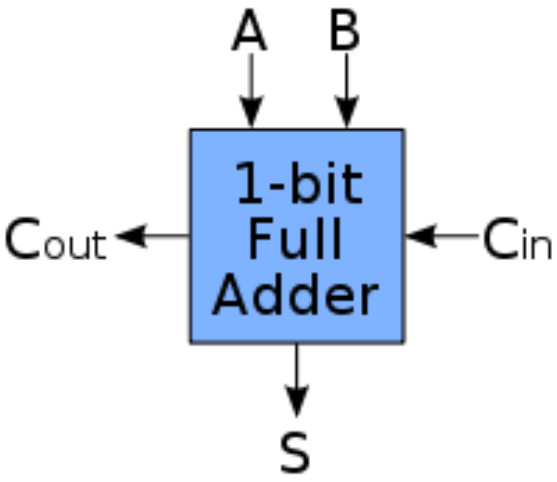
```
carry → 1
        1101 (a)
        1111 (b)
+       0010 (c)
----------------
        0000
```



Critical paths primarily require additions

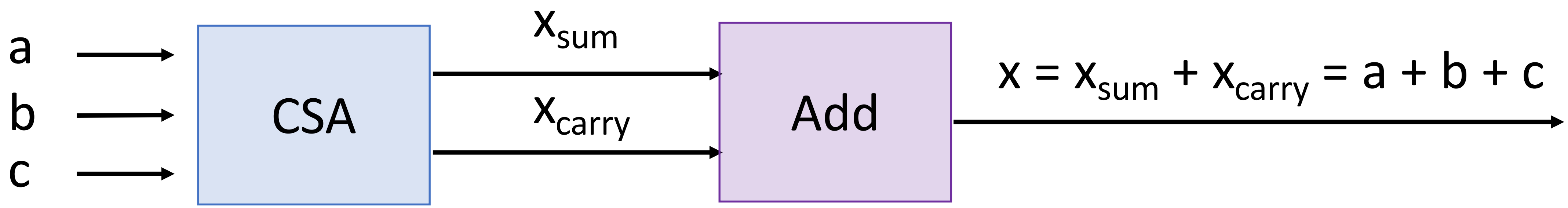
- The fastest adder is a carry-save adder (CSA)
  - Eliminates carry propagation, requiring $O(1)$ delay
  - Stores numbers in CSA form or redundant binary form

```
     1101  (a)
     1111  (b)
 +   0010  (c)
 ----------------
     0000
    11110
```



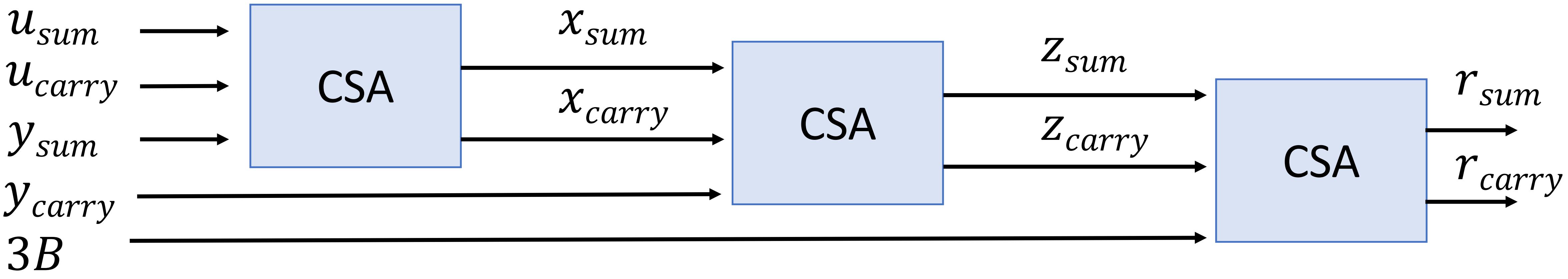
Critical paths primarily require additions

- 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



Two-bit PM with CSAs

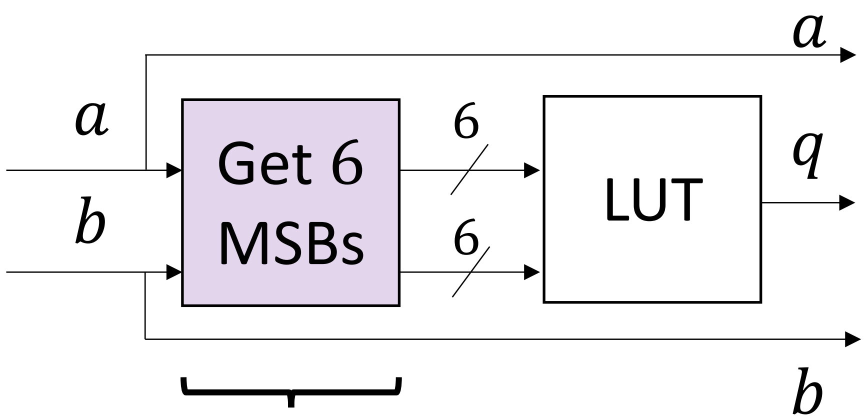
$$\frac{u + y + 3B}{4}$$



Critical path delay is 3 CSA delays

Euclid with CSAs

Compute $q \leq \lfloor \frac{a}{b} \rfloor$ ⟶ Compute $q * b$ ⟶ Compute a $- q * b$

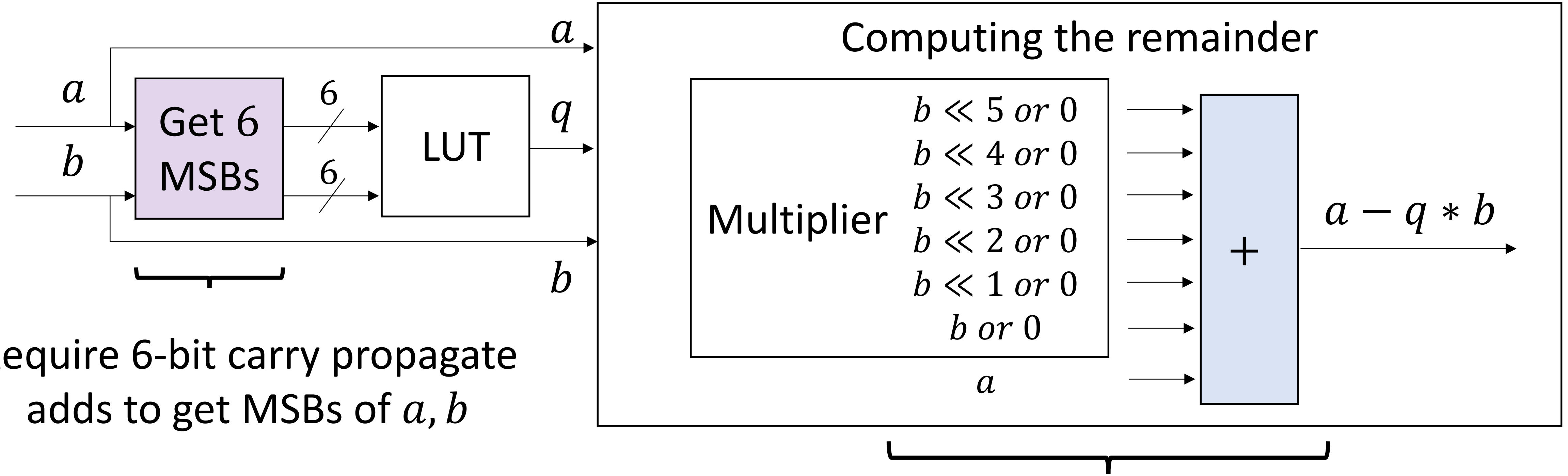


Require 6-bit carry propagate adds to get MSBs of a, b

$\lceil \log_2(6) \rceil + 1 = 3$ CSA delays

Euclid with CSAs

Compute $q \leq \lfloor \frac{a}{b} \rfloor$ $\longrightarrow$ Compute $q * b$ $\longrightarrow$ Compute $a - q * b$



Require 6-bit carry propagate adds to get MSBs of a, b

$\lfloor \log_2(6) \rfloor + 1 = 3$ CSA delays

Need to add 14 values with CSAs

$\approx \lfloor \log_{3/2}(14) \rfloor = 6$ CSA delays

Two-bit PM is a faster starting point

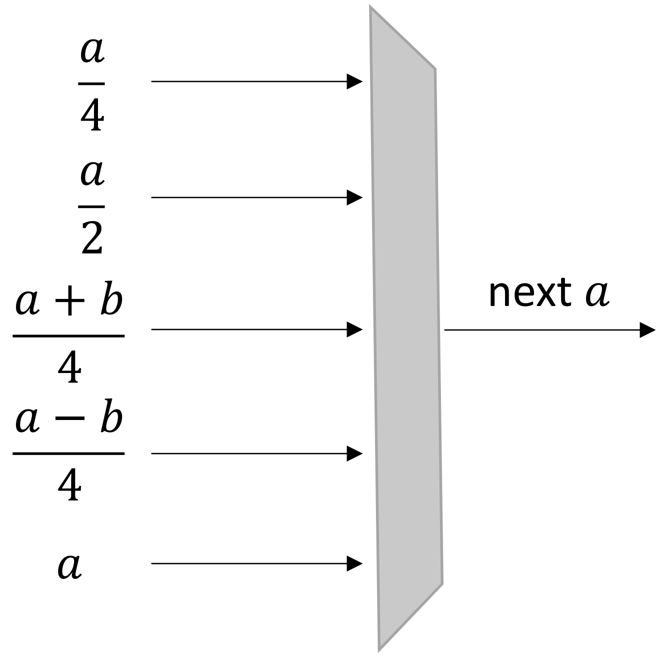
- Two-bit PM critical path delay estimate is 3X shorter than Euclid’s
- Two-bit PM iteration counts are at most 2X higher than Euclid’s

Two-bit PM with carry-save adders is the more promising starting point for hardware in the average and the worst-case.

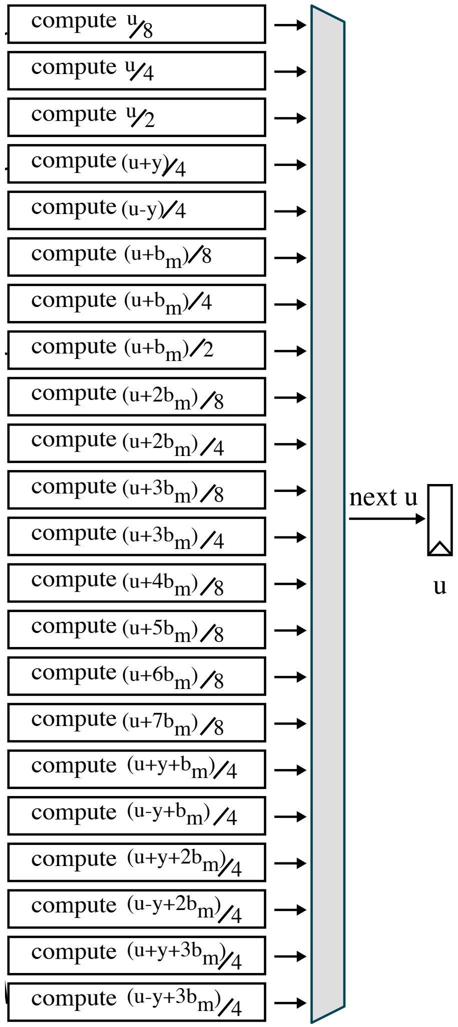
We build from the two-bit PM

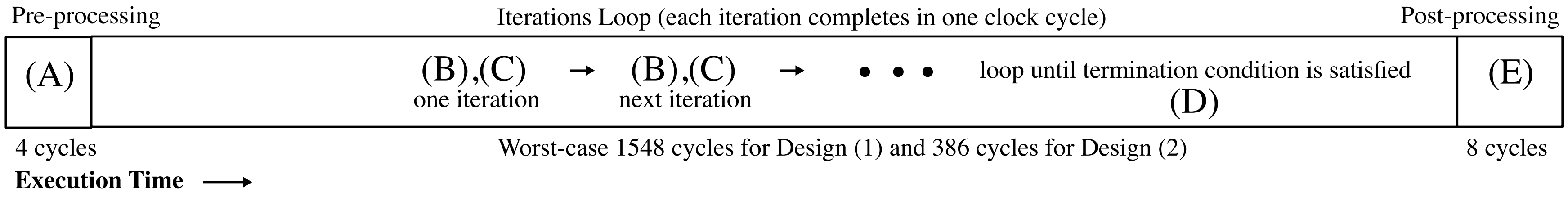
Two-bit Plus-Minus (PM)

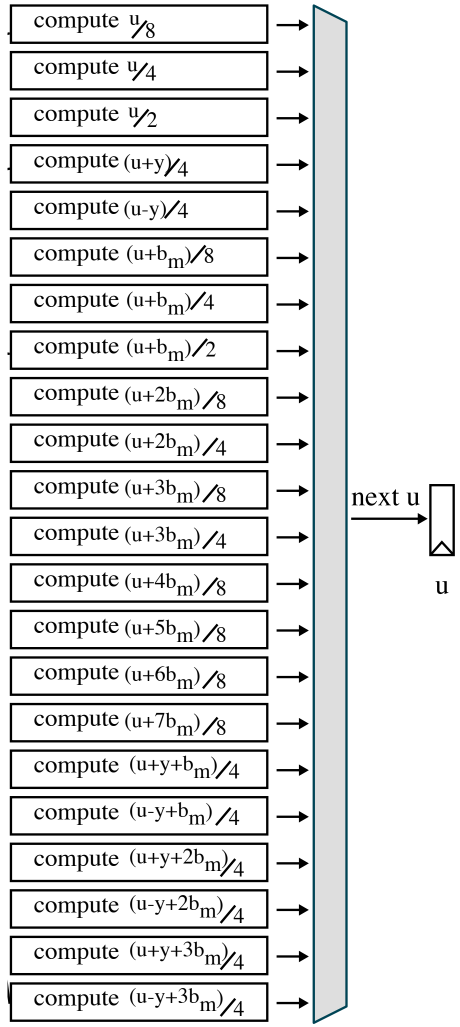
a	b	Operation
27	2	original a, b
27	1	b / 2
7	1	(a + b) / 4
2	1	(a + b) / 4
1	1	a / 2
1	0	(a – b) / 4

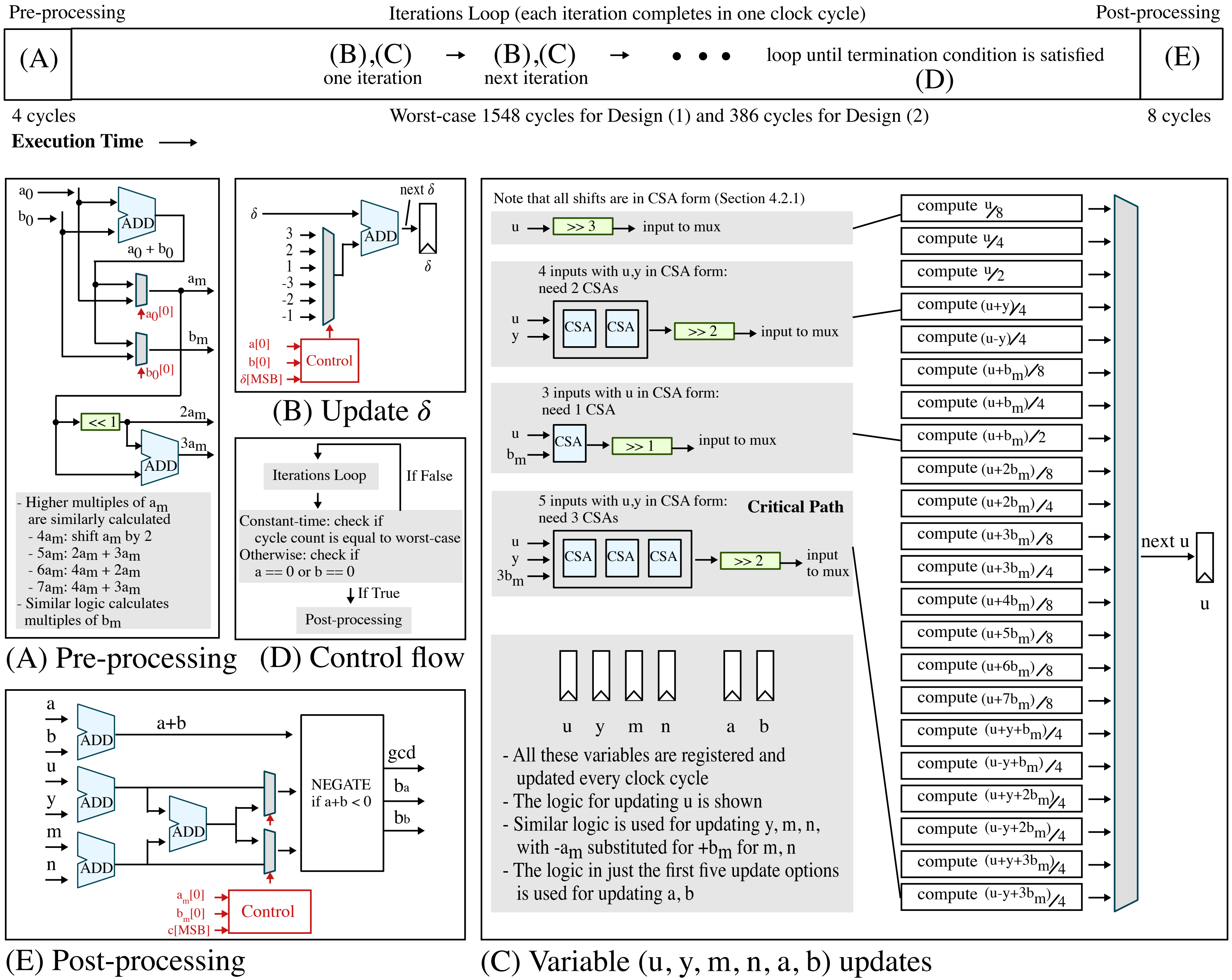


We extend two-bit PM for XGCD

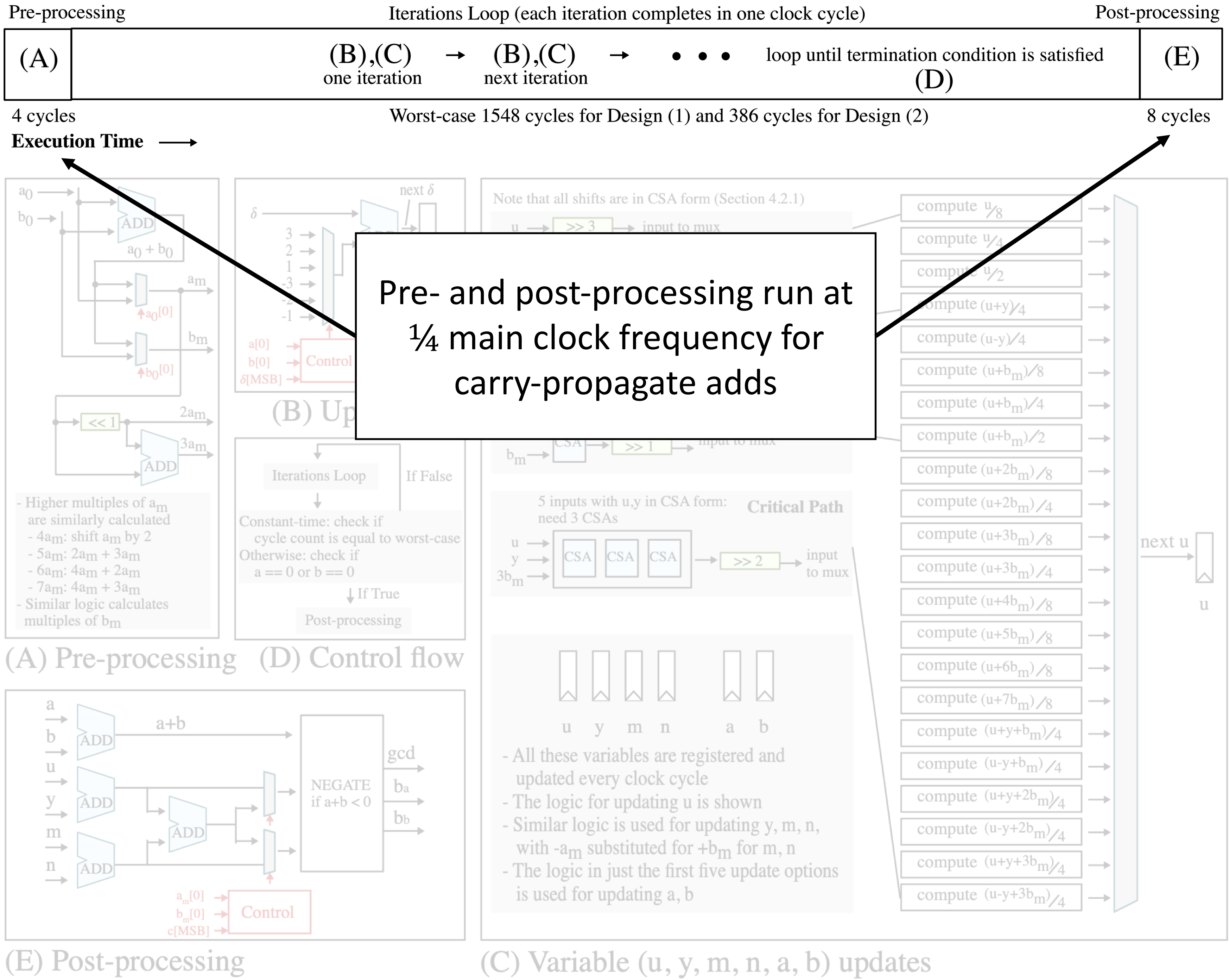



Pre-processing
Iterations Loop (each iteration completes in one clock cycle)
Post-processing
(A)
(B),(C)
one iteration
(B),(C)
next iteration
loop until termination condition is satisfied
(D)
(E)
4 cycles
Worst-case 1548 cycles for Design (1) and 386 cycles for Design (2)
8 cycles
Execution Time

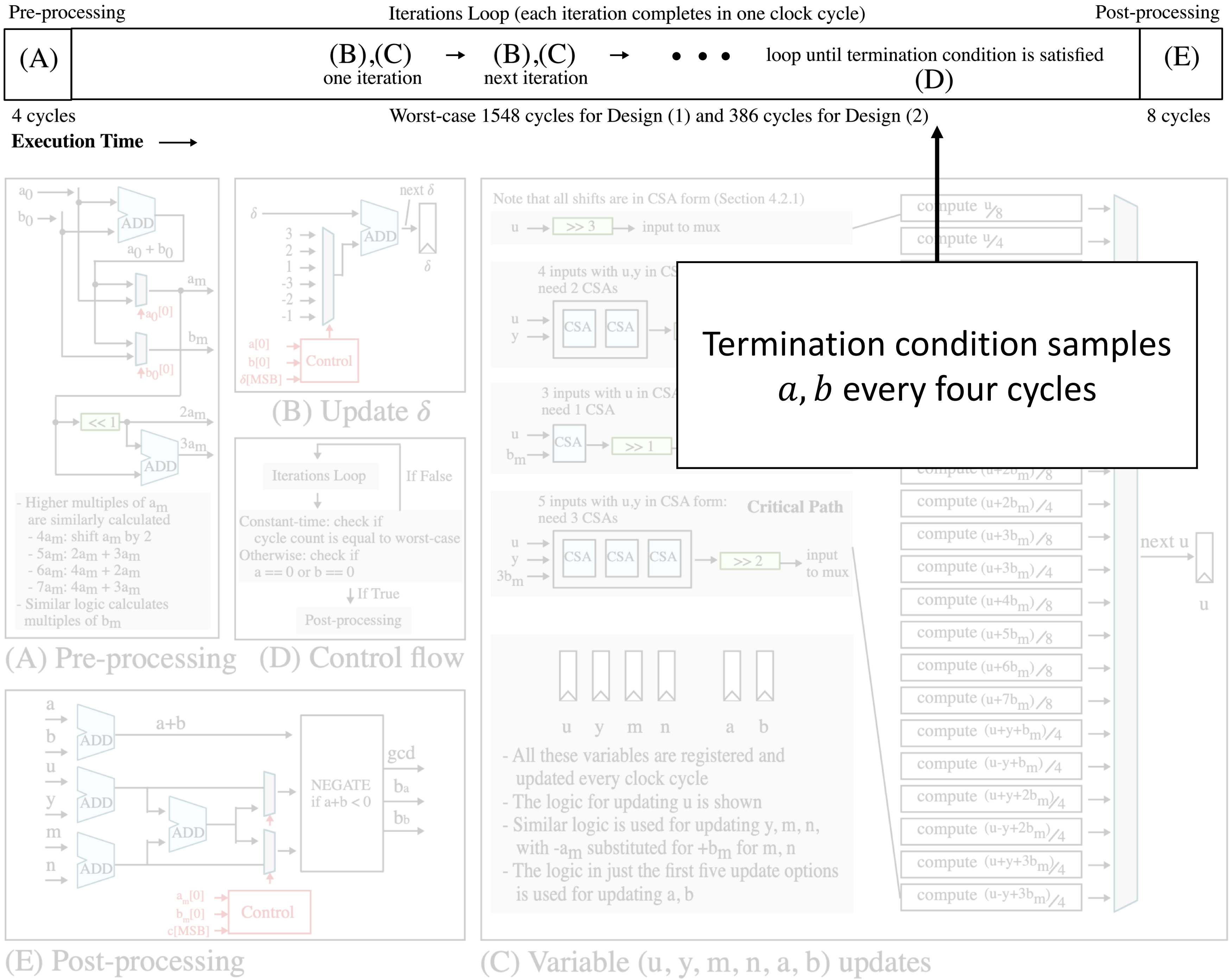

compute u/8
compute u/4
compute u/2
compute (u+y)/4
compute (u-y)/4
compute (u+b_m)/8
compute (u+b_m)/4
compute (u+b_m)/2
compute (u+2b_m)/8
compute (u+2b_m)/4
compute (u+3b_m)/8
compute (u+3b_m)/4
compute (u+4b_m)/8
compute (u+5b_m)/8
compute (u+6b_m)/8
compute (u+7b_m)/8
compute (u+y+b_m)/4
compute (u-y+b_m)/4
compute (u+y+2b_m)/4
compute (u-y+2b_m)/4
compute (u+y+3b_m)/4
compute (u-y+3b_m)/4
next u
u

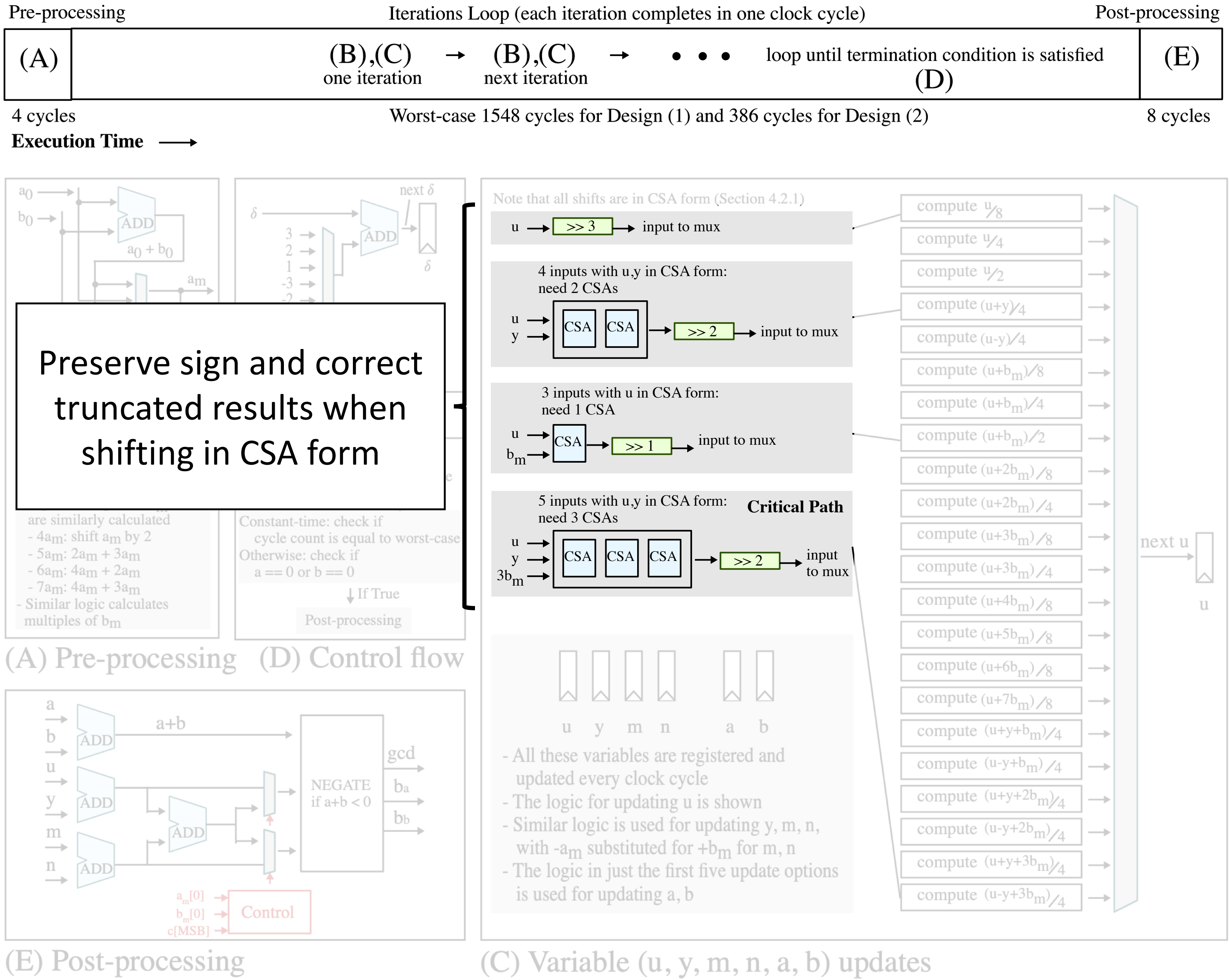


Pre-processing
Iterations Loop (each iteration completes in one clock cycle)
Post-processing
(A)
(B),(C)
one iteration
(B),(C)
next iteration
loop until termination condition is satisfied
(D)
(E)
4 cycles
Worst-case 1548 cycles for Design (1) and 386 cycles for Design (2)
8 cycles
Execution Time
a0
b0
ADD
a0 + b0
am
a0[0]
bm
b0[0]
2am
<< 1
3am
ADD
- Higher multiples of am are similarly calculated
- 4am: shift am by 2
- 5am: 2am + 3am
- 6am: 4am + 2am
- 7am: 4am + 3am
- Similar logic calculates multiples of bm
(A) Pre-processing
next δ
δ
3
2
1
-3
-2
-1
ADD
a[0]
b[0]
δ[MSB]
Control
(B) Update δ
Iterations Loop
If False
Constant-time: check if cycle count is equal to worst-case
Otherwise: check if a == 0 or b == 0
If True
Post-processing
(D) Control flow
Note that all shifts are in CSA form (Section 4.2.1)
u
>> 3
input to mux
4 inputs with u,y in CSA form: need 2 CSAs
u
y
CSA
CSA
>> 2
input to mux
3 inputs with u in CSA form: need 1 CSA
u
bm
CSA
>> 1
input to mux
5 inputs with u,y in CSA form: need 3 CSAs
Critical Path
u
y
3bm
CSA
CSA
CSA
>> 2
input to mux
u y m n a b
- All these variables are registered and updated every clock cycle
- The logic for updating u is shown
- Similar logic is used for updating y, m, n, with -am substituted for +bm for m, n
- The logic in just the first five update options is used for updating a, b
compute u/8
compute u/4
compute u/2
compute (u+y)/4
compute (u-y)/4
compute (u+bm)/8
compute (u+bm)/4
compute (u+bm)/2
compute (u+2bm)/8
compute (u+2bm)/4
compute (u+3bm)/8
compute (u+3bm)/4
compute (u+4bm)/8
compute (u+5bm)/8
compute (u+6bm)/8
compute (u+7bm)/8
compute (u+y+bm)/4
compute (u-y+bm)/4
compute (u+y+2bm)/4
compute (u-y+2bm)/4
compute (u+y+3bm)/4
compute (u-y+3bm)/4
next u
u
a
b
u
y
m
n
ADD
ADD
ADD
ADD
a+b
NEGATE if a+b < 0
gcd
ba
bb
am[0]
bm[0]
c[MSB]
Control
(E) Post-processing
(C) Variable (u, y, m, n, a, b) updates

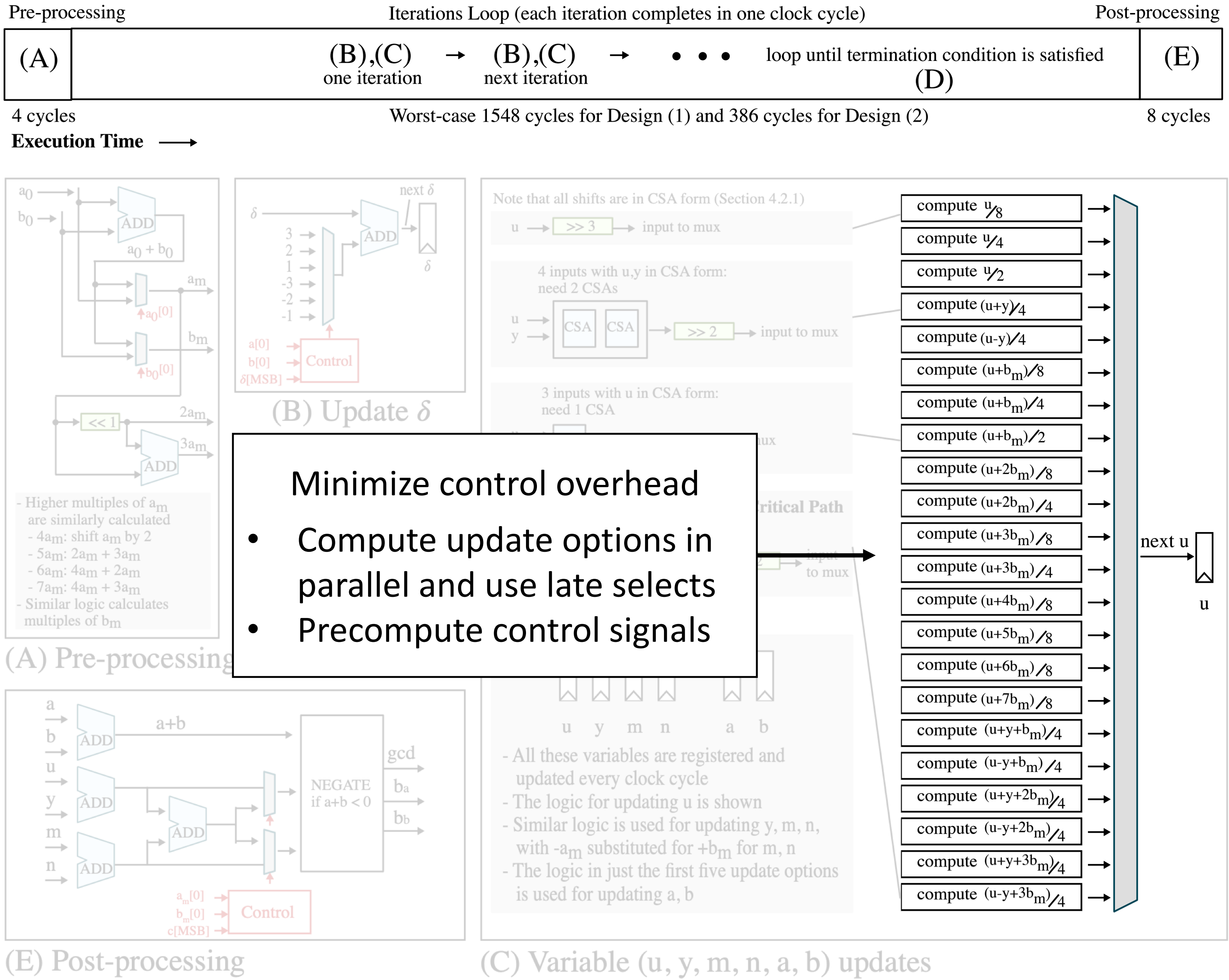


Pre-processing
Iterations Loop (each iteration completes in one clock cycle)
Post-processing
(A)
(B),(C)
one iteration
(B),(C)
next iteration
loop until termination condition is satisfied
(D)
(E)
4 cycles
Worst-case 1548 cycles for Design (1) and 386 cycles for Design (2)
8 cycles
Execution Time
Pre- and post-processing run at ¼ main clock frequency for carry-propagate adds
(A) Pre-processing
(D) Control flow
(E) Post-processing
(C) Variable (u, y, m, n, a, b) updates
Note that all shifts are in CSA form (Section 4.2.1)
Critical Path
5 inputs with u,y in CSA form: need 3 CSAs
- Higher multiples of am are similarly calculated
- Similar logic calculates multiples of bm
Iterations Loop
If False
Constant-time: check if cycle count is equal to worst-case
Otherwise: check if a == 0 or b == 0
If True
Post-processing
NEGATE if a+b < 0
Control
- All these variables are registered and updated every clock cycle
- The logic for updating u is shown
- Similar logic is used for updating y, m, n, with -am substituted for +bm for m, n
- The logic in just the first five update options is used for updating a, b
next u


Pre-processing
Iterations Loop (each iteration completes in one clock cycle)
Post-processing
(A)
(B),(C)
one iteration
(B),(C)
next iteration
loop until termination condition is satisfied
(D)
(E)
4 cycles
Worst-case 1548 cycles for Design (1) and 386 cycles for Design (2)
8 cycles
Execution Time
Termination condition samples
a, b every four cycles
(A) Pre-processing
(B) Update δ
(D) Control flow
(E) Post-processing
(C) Variable (u, y, m, n, a, b) updates
Note that all shifts are in CSA form (Section 4.2.1)
Iterations Loop
If False
Constant-time: check if cycle count is equal to worst-case
Otherwise: check if a == 0 or b == 0
If True
Post-processing
Critical Path
- Higher multiples of am are similarly calculated
- All these variables are registered and updated every clock cycle
- The logic for updating u is shown
- Similar logic is used for updating y, m, n, with -am substituted for +bm for m, n
- The logic in just the first five update options is used for updating a, b
NEGATE if a+b < 0
Control



Pre-processing
Iterations Loop (each iteration completes in one clock cycle)
Post-processing
(A)
(B),(C) one iteration
(B),(C) next iteration
loop until termination condition is satisfied (D)
(E)
4 cycles
Worst-case 1548 cycles for Design (1) and 386 cycles for Design (2)
8 cycles
Execution Time
Preserve sign and correct truncated results when shifting in CSA form
Note that all shifts are in CSA form (Section 4.2.1)
u >> 3 input to mux
4 inputs with u,y in CSA form: need 2 CSAs
u y CSA CSA >> 2 input to mux
3 inputs with u in CSA form: need 1 CSA
u bm CSA >> 1 input to mux
5 inputs with u,y in CSA form: need 3 CSAs
Critical Path
u y 3bm CSA CSA CSA >> 2 input to mux
compute u/8
compute u/4
compute u/2
compute (u+y)/4
compute (u-y)/4
compute (u+bm)/8
compute (u+bm)/4
compute (u+bm)/2
compute (u+2bm)/8
compute (u+2bm)/4
compute (u+3bm)/8
compute (u+3bm)/4
compute (u+4bm)/8
compute (u+5bm)/8
compute (u+6bm)/8
compute (u+7bm)/8
compute (u+y+bm)/4
compute (u-y+bm)/4
compute (u+y+2bm)/4
compute (u-y+2bm)/4
compute (u+y+3bm)/4
compute (u-y+3bm)/4
next u
u
u y m n a b
- All these variables are registered and updated every clock cycle
- The logic for updating u is shown
- Similar logic is used for updating y, m, n, with -am substituted for +bm for m, n
- The logic in just the first five update options is used for updating a, b
are similarly calculated
- 4am: shift am by 2
- 5am: 2am + 3am
- 6am: 4am + 2am
- 7am: 4am + 3am
- Similar logic calculates multiples of bm
Constant-time: check if cycle count is equal to worst-case
Otherwise: check if a == 0 or b == 0
If True
Post-processing
(A) Pre-processing
(D) Control flow
a b u y m n
ADD
a+b
NEGATE if a+b < 0
gcd
ba
bb
Control
(E) Post-processing
(C) Variable (u, y, m, n, a, b) updates


Pre-processing
Iterations Loop (each iteration completes in one clock cycle)
Post-processing
(A)
(B),(C)
one iteration
(B),(C)
next iteration
loop until termination condition is satisfied
(D)
(E)
4 cycles
Worst-case 1548 cycles for Design (1) and 386 cycles for Design (2)
8 cycles
Execution Time
Minimize control overhead
• Compute update options in parallel and use late selects
• Precompute control signals
compute u/8
compute u/4
compute u/2
compute (u+y)/4
compute (u-y)/4
compute (u+bm)/8
compute (u+bm)/4
compute (u+bm)/2
compute (u+2bm)/8
compute (u+2bm)/4
compute (u+3bm)/8
compute (u+3bm)/4
compute (u+4bm)/8
compute (u+5bm)/8
compute (u+6bm)/8
compute (u+7bm)/8
compute (u+y+bm)/4
compute (u-y+bm)/4
compute (u+y+2bm)/4
compute (u-y+2bm)/4
compute (u+y+3bm)/4
compute (u-y+3bm)/4
next u
u
Note that all shifts are in CSA form (Section 4.2.1)
4 inputs with u,y in CSA form: need 2 CSAs
3 inputs with u in CSA form: need 1 CSA
input to mux
- Higher multiples of am are similarly calculated
- Similar logic calculates multiples of bm
- All these variables are registered and updated every clock cycle
- The logic for updating u is shown
- Similar logic is used for updating y, m, n, with -am substituted for +bm for m, n
- The logic in just the first five update options is used for updating a, b
NEGATE if a+b < 0
(A) Pre-processing
(B) Update δ
(E) Post-processing
(C) Variable (u, y, m, n, a, b) updates

Critical Path in 16nm

1024 bits | 255 bits

Operation	Design (1) Delay (ns)	Design (1) FO4 Inv Delay	Design (2) Delay (ns)	Design (2) FO4 Inv Delay
Local clock gating	0.035	3.9	0.018	2
DFF clk to Q	0.040	4.4	0.045	5
Inverter	0	0	0.007	0.8
Add $u + y$: CSA 1	0.039	4.3	0.018	2
Add $u + y$: CSA 2	0.039	4.3	0.031	3.4
Buffer	0	0	0.013	1.4
Add $u + y + 2b_m$: CSA	0.034	3.8	0.030	3.3
Shift in CSA form	0.018	2	0.015	1.7
Late select multiplexers	0.018	2	0.018	2
Precomputing control	0.022	2.4	0.027	3
Total	0.257	28.6	0.220	24.4

Is three-bit PM faster in hardware?

1024 bits

Max factor of two reduction when a *or* b is *even*	Max factor of two reduction when a *and* b are *odd*	Average Number of Cycles	Cycle Time (ns)	XGCD execution time (ns)	ASIC area (mm^2)
2	2	2210	0.193	427	0.16
4	2	1845	0.218	402	0.21
8	2	1740	0.251	437	0.35
2	4	1450	0.234	339	0.22
4	4	1211	0.247	299	0.28
8	**4**	**1143**	**0.257**	**294**	**0.41**
2	8	1091	0.297	324	0.27
4	8	972	0.320	311	0.33
8	8	937	0.330	309	0.47

Yes, three-bit PM has lowest average execution time

Is three-bit PM faster in hardware?

1024 bits

Max factor of two reduction when *a or b* is *even*	Max factor of two reduction when *a and b* are *odd*	Average Number of Cycles	Cycle Time (ns)	XGCD execution time (ns)	ASIC area (mm^2)
2	2	2210	0.193	427	0.16
4	2	1845	0.218	402	0.21
8	2	1740	0.251	437	0.35
2	4	1450	0.234	339	0.22
4	4	1211	0.247	299	0.28
8	**4**	**1143**	**0.257**	**294**	**0.41**
2	8	1091	0.297	324	0.27
4	8	972	0.320	311	0.33
8	8	937	0.330	309	0.47

For constant-time

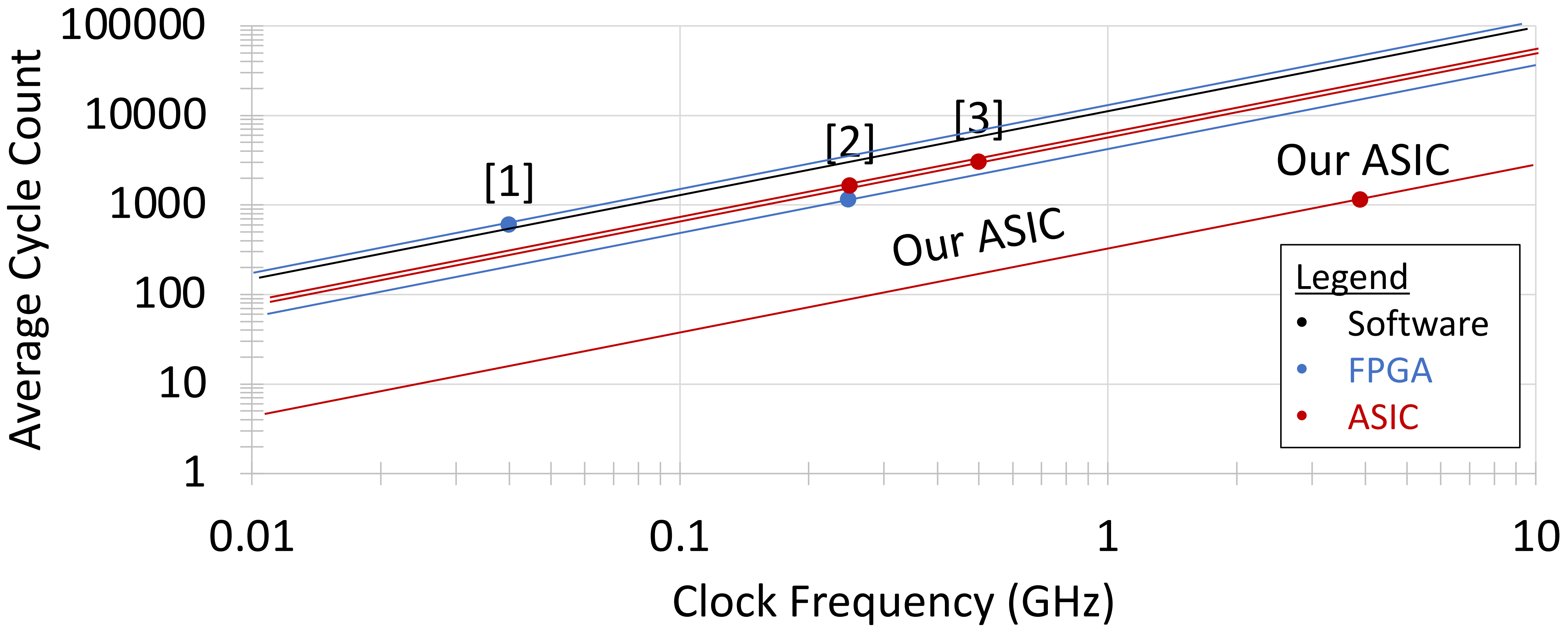
No, all these cases have same worst-case execution time.

Constant-time and polynomial extensions

- Constant-time evaluation always runs worst-case number of cycles
  - Algorithm keeps dividing 0 by 2 when run for more cycles
  - Luckily, CSA form makes it unclear when a, b are 0

- Polynomial XGCD maps integer operations to polynomial ones
  - Reducing factors of 2 $\Rightarrow$ Reducing factors of x
  - Checking evenness $\Rightarrow$ Checking divisibility by x
  - Comparing integers $\Rightarrow$ Comparing polynomial degrees

1024-bit Fast Average XGCD Comparisons

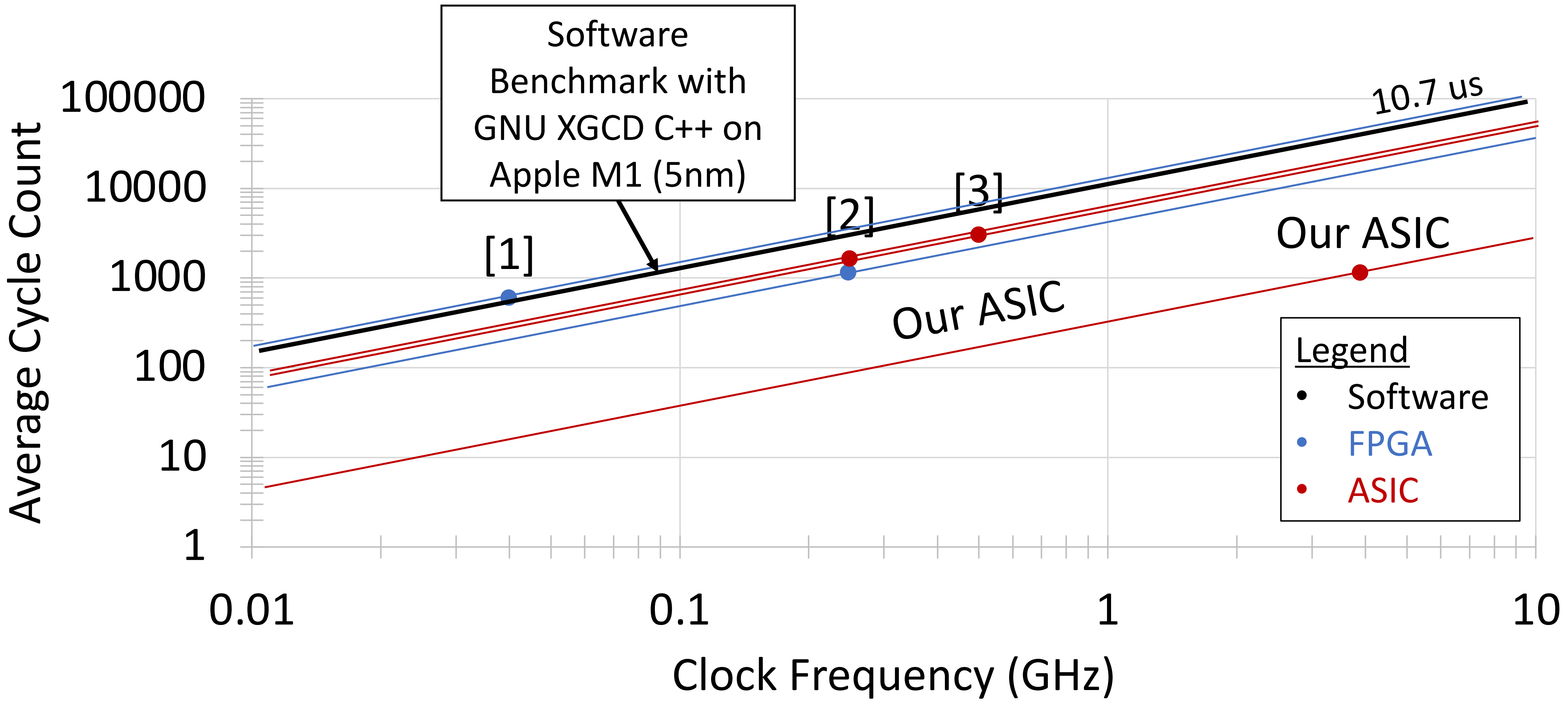


[1] Al-Haija et al. A comparative study up to 1024 bit euclid's gcd algorithm fpga implementation and synthesizing. ICEDSA 2016.

[2] Zhu et al. Low-latency architecture for the parallel extended GCD algorithm of large numbers. ISCAS 2021.

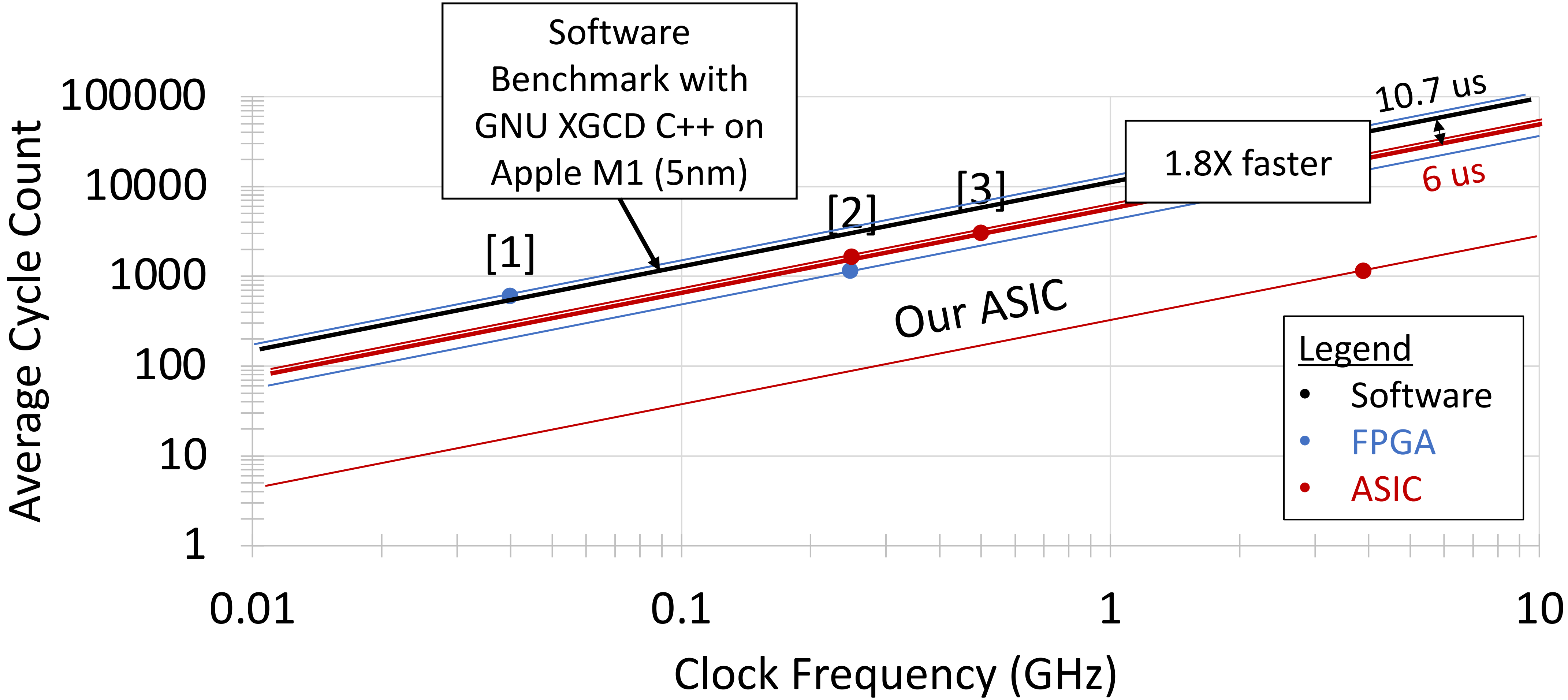
[3] Zhu et al. An efficient accelerator of the squaring for the verifiable delay function over a class group. APCCAS 2020.

1024-bit Fast Average XGCD Comparisons



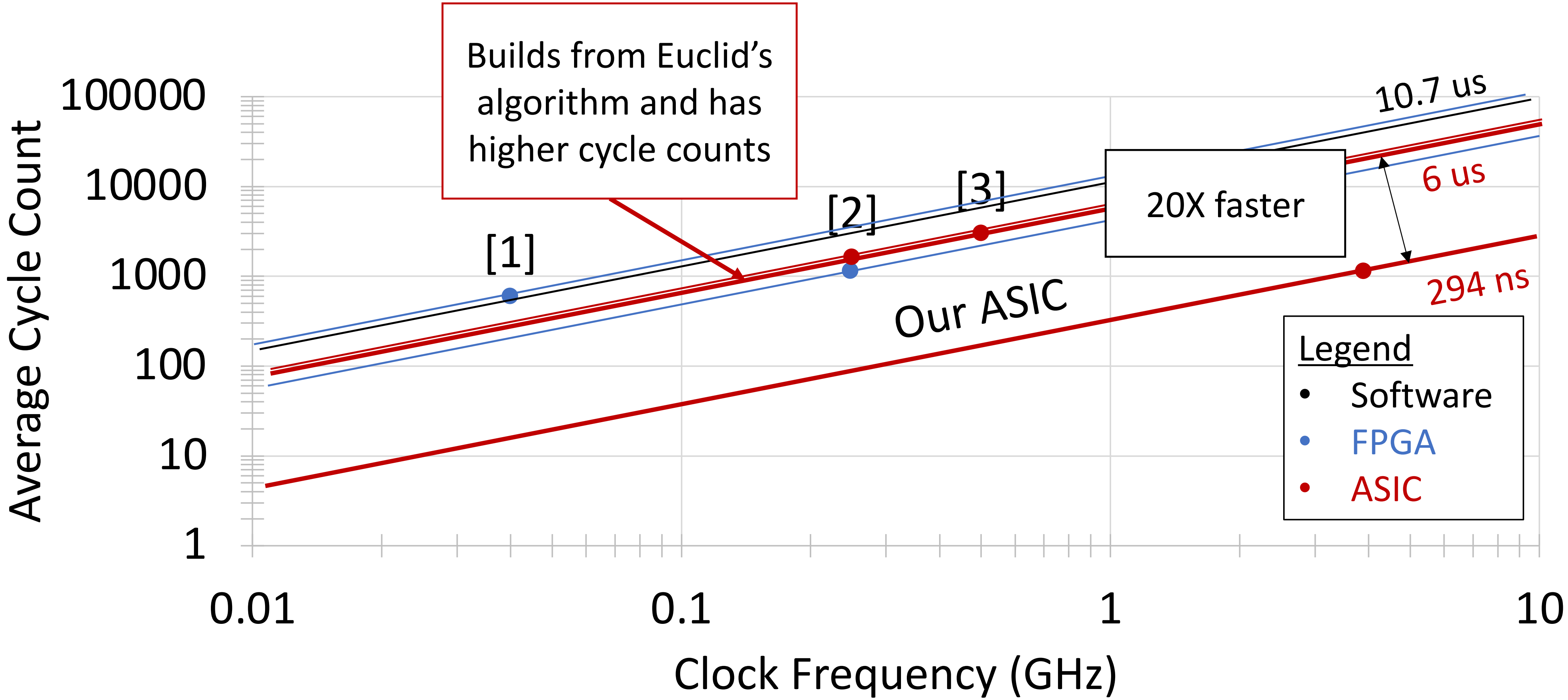
[1] Al-Haija et al. A comparative study up to 1024 bit euclid's gcd algorithm fpga implementation and synthesizing. ICEDSA 2016.
[2] Zhu et al. Low-latency architecture for the parallel extended GCD algorithm of large numbers. ISCAS 2021.
[3] Zhu et al. An efficient accelerator of the squaring for the verifiable delay function over a class group. APCCAS 2020.

1024-bit Fast Average XGCD Comparisons



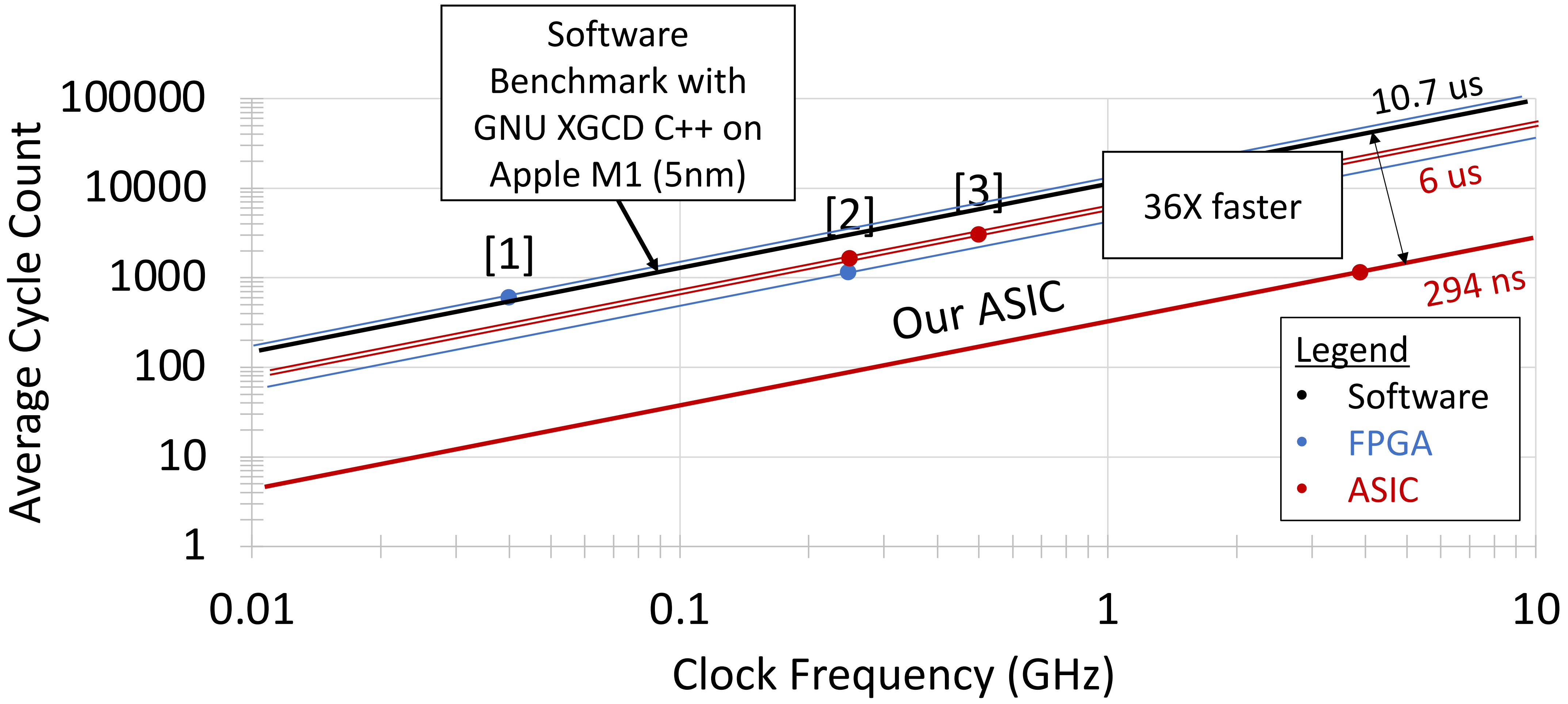
[1] Al-Haija et al. A comparative study up to 1024 bit euclid's gcd algorithm fpga implementation and synthesizing. ICEDSA 2016.
[2] Zhu et al. Low-latency architecture for the parallel extended GCD algorithm of large numbers. ISCAS 2021.
[3] Zhu et al. An efficient accelerator of the squaring for the verifiable delay function over a class group. APCCAS 2020.

1024-bit Fast Average XGCD Comparisons



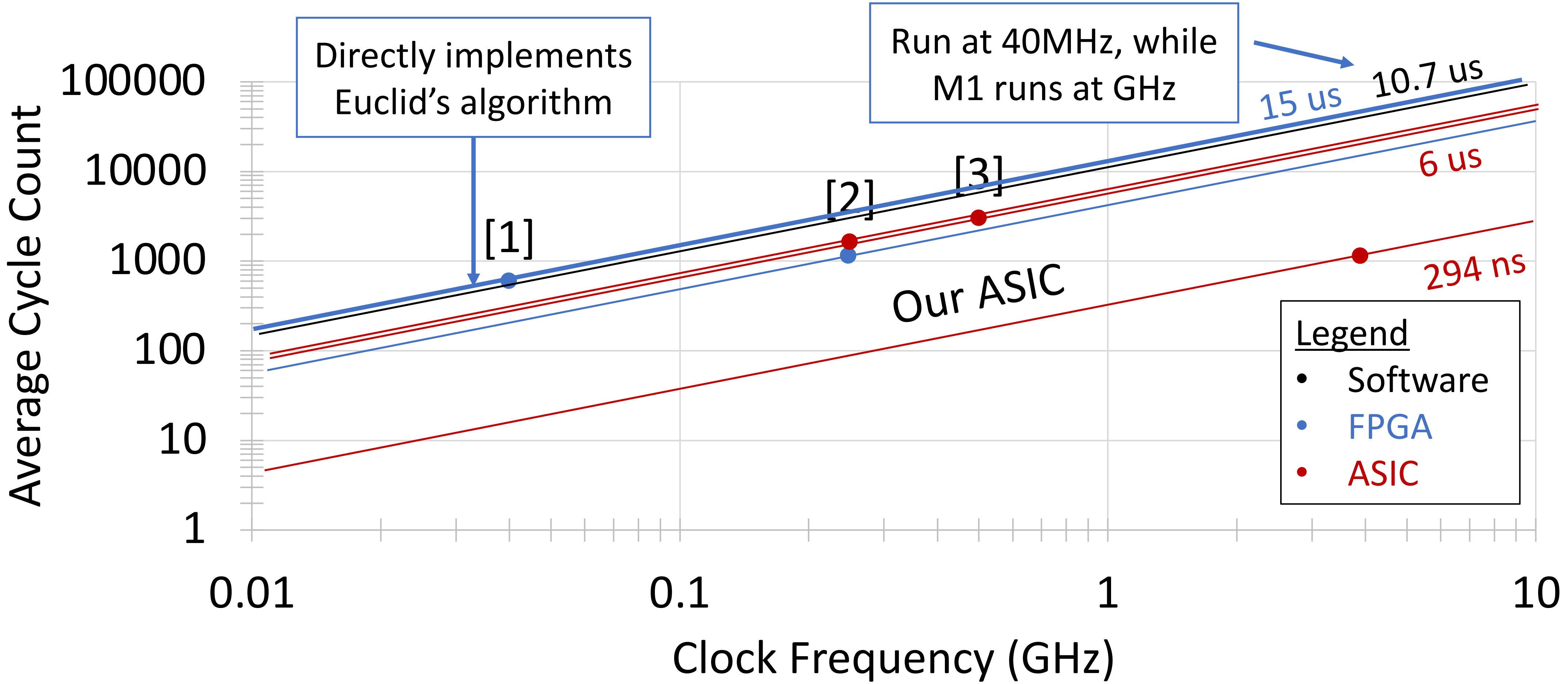
[1] Al-Haija et al. A comparative study up to 1024 bit euclid's gcd algorithm fpga implementation and synthesizing. ICEDSA 2016.
[2] Zhu et al. Low-latency architecture for the parallel extended GCD algorithm of large numbers. ISCAS 2021.
[3] Zhu et al. An efficient accelerator of the squaring for the verifiable delay function over a class group. APCCAS 2020.

1024-bit Fast Average XGCD Comparisons



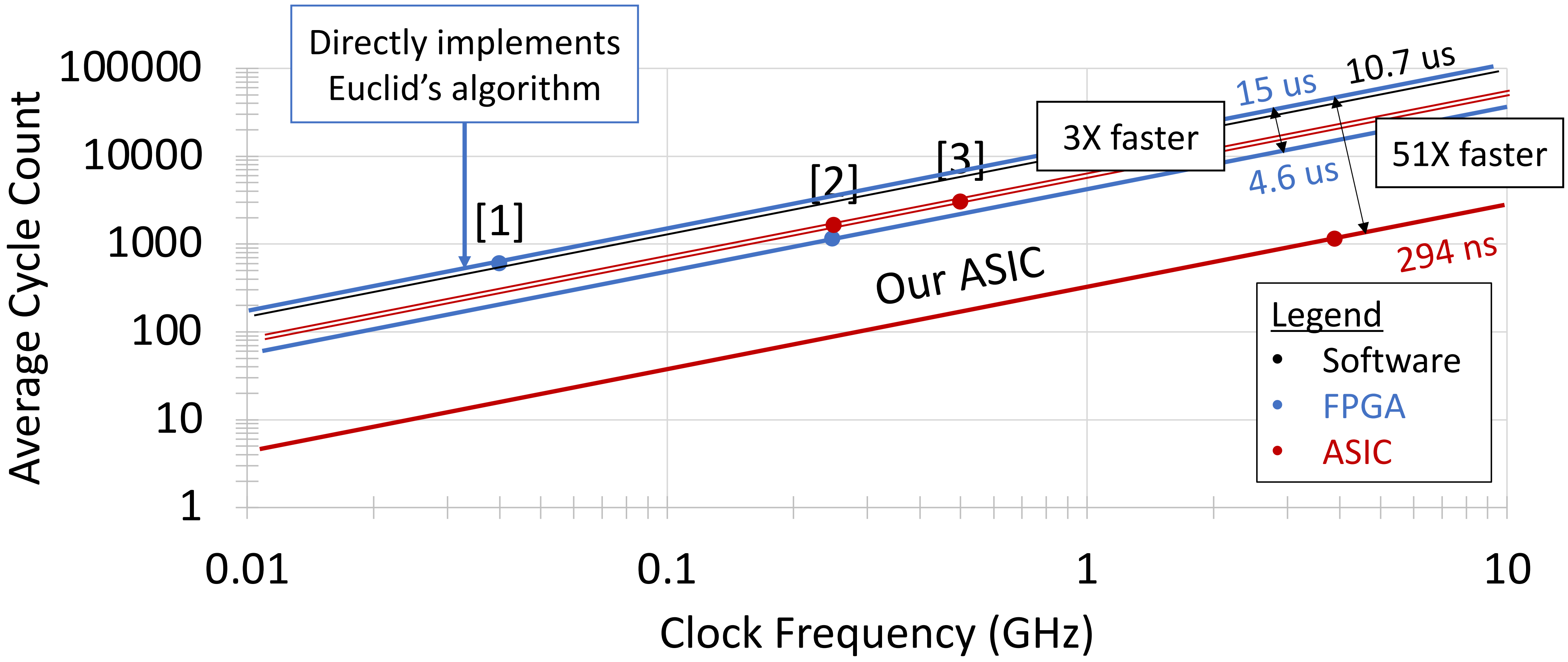
[1] Al-Haija et al. A comparative study up to 1024 bit euclid's gcd algorithm fpga implementation and synthesizing. ICEDSA 2016.
[2] Zhu et al. Low-latency architecture for the parallel extended GCD algorithm of large numbers. ISCAS 2021.
[3] Zhu et al. An efficient accelerator of the squaring for the verifiable delay function over a class group. APCCAS 2020.

1024-bit Fast Average XGCD Comparisons



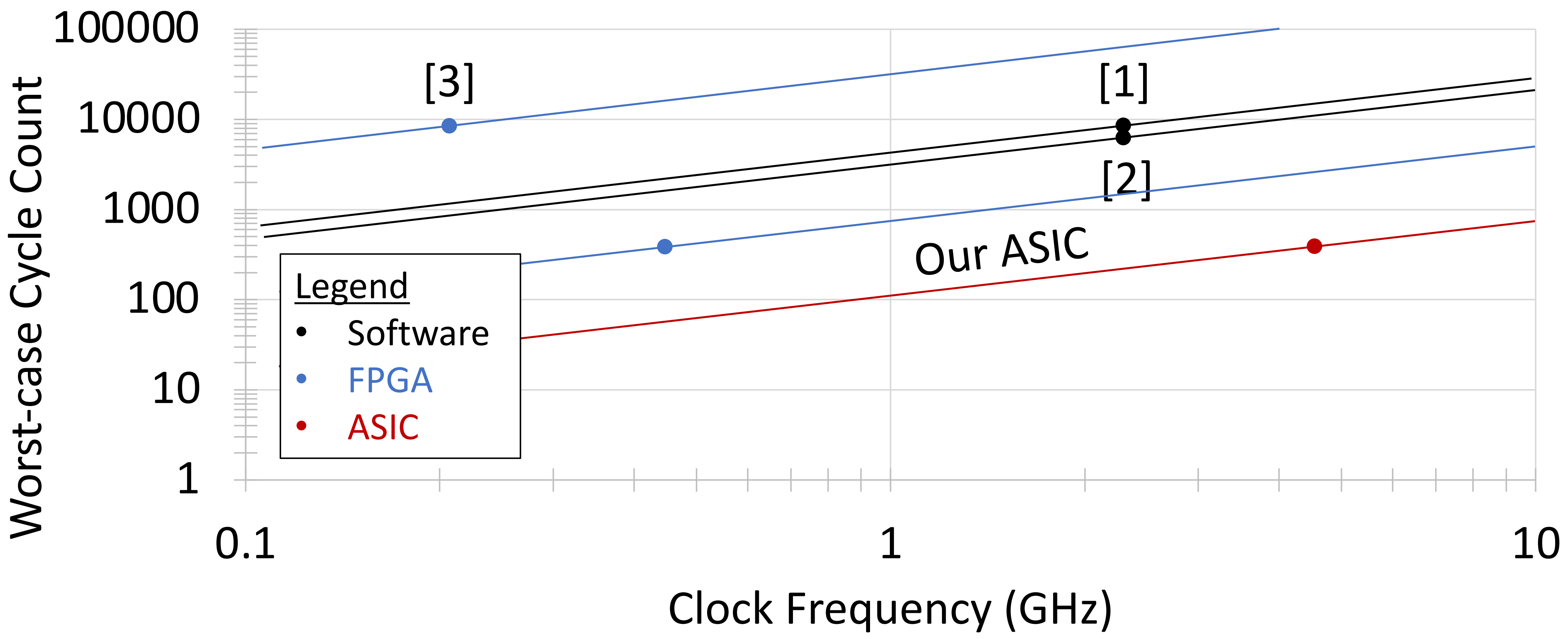
[1] Al-Haija et al. A comparative study up to 1024 bit euclid's gcd algorithm fpga implementation and synthesizing. ICEDSA 2016.
[2] Zhu et al. Low-latency architecture for the parallel extended GCD algorithm of large numbers. ISCAS 2021.
[3] Zhu et al. An efficient accelerator of the squaring for the verifiable delay function over a class group. APCCAS 2020.

1024-bit Fast Average XGCD Comparisons



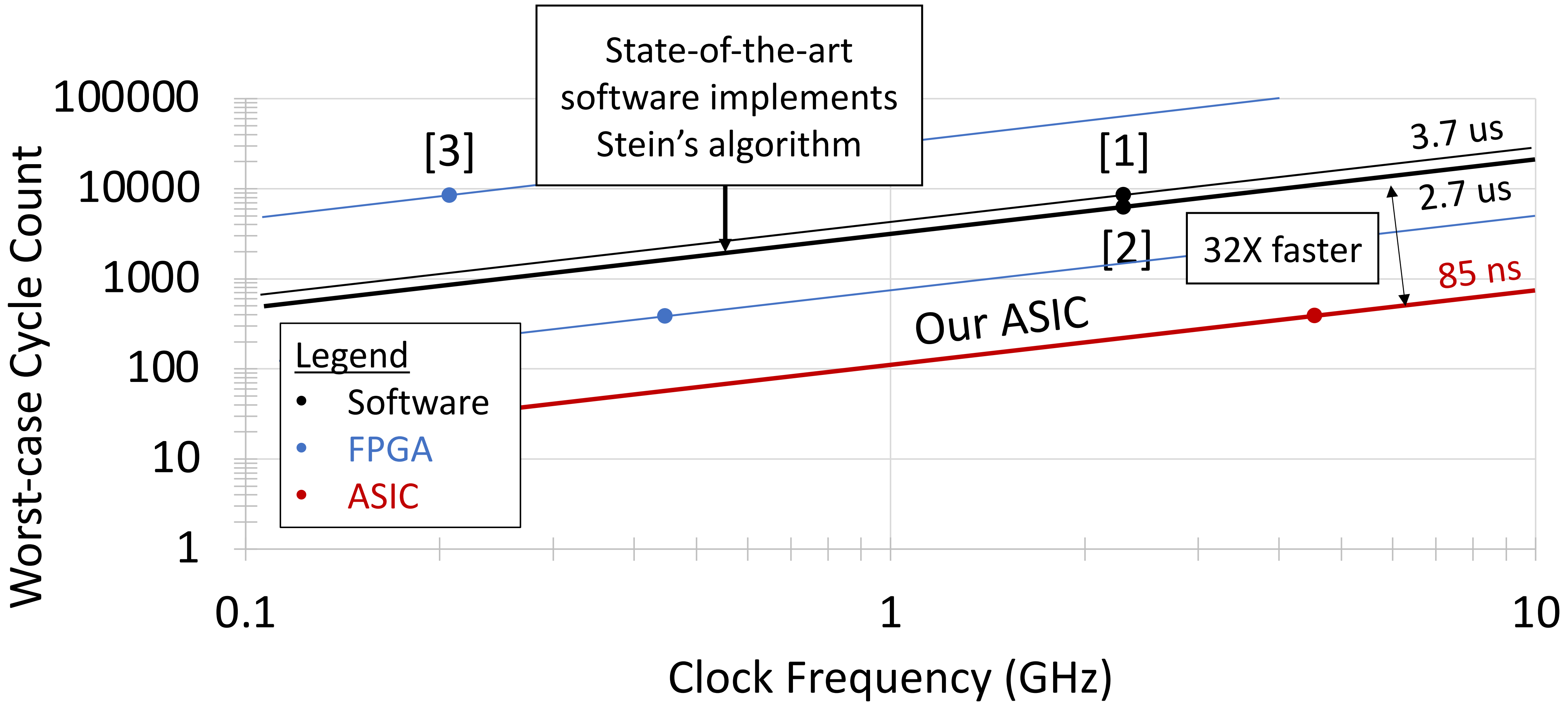
[1] Al-Haija et al. A comparative study up to 1024 bit euclid’s gcd algorithm fpga implementation and synthesizing. ICEDSA 2016.
[2] Zhu et al. Low-latency architecture for the parallel extended GCD algorithm of large numbers. ISCAS 2021.
[3] Zhu et al. An efficient accelerator of the squaring for the verifiable delay function over a class group. APCCAS 2020.

255-bit Constant-time XGCD Comparisons



[1] Bernstein and Yang. Fast constant-time gcd computation and modular inversion. CHES 2019.
[2] Pornin. Optimized binary gcd for modular inversion. Cryptology ePrint Archive 2020.
[3] Deshpande et al. Modular inverse for integers using fast constant time gcd algorithm and its applications. FPL 2021.

255-bit Constant-time XGCD Comparisons

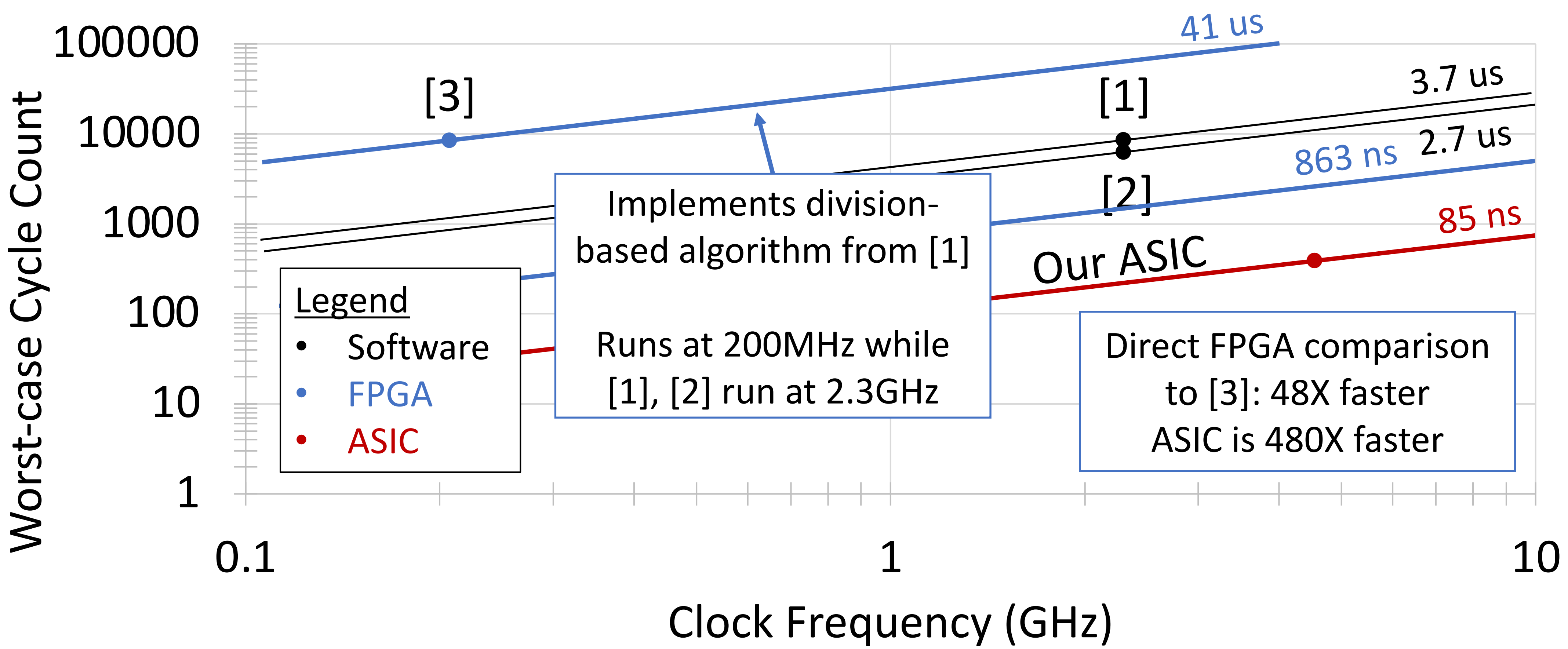


[1] Bernstein and Yang. Fast constant-time gcd computation and modular inversion. CHES 2019.

[2] Pornin. Optimized binary gcd for modular inversion. Cryptology ePrint Archive 2020.

[3] Deshpande et al. Modular inverse for integers using fast constant time gcd algorithm and its applications. FPL 2021.

255-bit Constant-time XGCD Comparisons



[1] Bernstein and Yang. Fast constant-time gcd computation and modular inversion. CHES 2019.

[2] Pornin. Optimized binary gcd for modular inversion. Cryptology ePrint Archive 2020.

[3] Deshpande et al. Modular inverse for integers using fast constant time gcd algorithm and its applications. FPL 2021.

Takeaways

- XGCD is critical for recent cryptographic applications
  - Two-bit PM + CSAs are more promising for hardware
- This approach gives order-of-magnitude better performance
  - 30 – 40X faster than software
  - 8X faster than state-of-the-art ASIC and first constant-time ASIC
- We plan to tape out these designs in GF12 in September