A 3.25 GHz Large-Integer Extended GCD Accelerator in 12 nm

Kavya Sreedhar
Stanford University
Stanford, USA
skavya@stanford.edu

Gedeon Nyengele
Stanford University
Stanford, USA
nyengele@stanford.edu

Mark Horowitz
Stanford University
Stanford, USA
horowitz@ee.stanford.edu

Christopher Torng
University of Southern California
Los Angeles, USA
ctorng@usc.edu

***Abstract*—Large-integer extended GCD (XGCD) is a critical operation in cryptography applications such as new blockchains and modular inversion. We present the first ASIC for XGCD, which is 23× faster than state-of-the-art software, 18× faster than prior hardware simulations, and up to 303× faster than prior chips for modular inversion. These performance gains come from careful circuit and logic optimizations to avoid long carry propagation and to hide control signal delays. Our chip computes XGCD with 255-bit integer inputs in 87 ns and XGCD with 512-bit integer inputs in 176 ns on average. Our chip can be configured for constant-time execution and computes constant-time 255-bit XGCD in 119 ns and constant-time 512-bit XGCD in 239 ns.**

***Index Terms*—extended GCD, modular inversion, constant time, carry save**

I. Introduction

The extended greatest common divisor (XGCD) computation finds Bézout coefficients u, m such that $u * a + m * b = GCD(a, b)$. XGCD has many applications in cryptography, including modular inversion [1], RSA [2], elliptic curve cryptography [3], blockchains [4], and ElGamal encryption [5]. In new blockchain protocols and newer efficient approaches for constant-time modular inversion, XGCD requires over 90% of the computation time [6]. As a result, recent papers have begun exploring fast hardware designs for large-integer XGCD [6]–[9], but there are no fabricated chips yet in the literature.

This paper demonstrates the performance advantage of our carry-save-adder (CSA) subtraction-based XGCD algorithm [6] and presents the first fabricated ASIC for XGCD. Using a CSA-based approach results in a fast cycle time, which is independent of input bitwidth: our chip, built in GlobalFoundries 12 nm FinFET technology, runs at 3.25 GHz for 255-bit and 512-bit inputs when driven with a 0.9 V power supply. Using a pure CSA-based approach presents several challenges: we cannot quickly compare values in carry-save (CS) form, the time needed to drive our control signals across very wide, 255 to 512-bit datapaths is a large fraction of our cycle time, and shifting CS values requires care to preserve data values. This paper extends our prior work [6] to overcome these challenges, by optimizing large-integer approximations for the control flow, performing area-performance tradeoffs with control logic optimizations, and providing circuitry for accurately and efficiently shifting in CS form. With these optimizations, our chip is 23× faster than state-of-the-art software, 18× faster than prior hardware simulations, and up to 303× faster than prior chips for modular inversion.

This work was supported by the DSSoC DARPA program and the Stanford AHA Agile Hardware Center and Affiliates Program. Kavya Sreedhar was supported by a graduate fellowship award as a Knight-Hennessy Scholar at Stanford University.

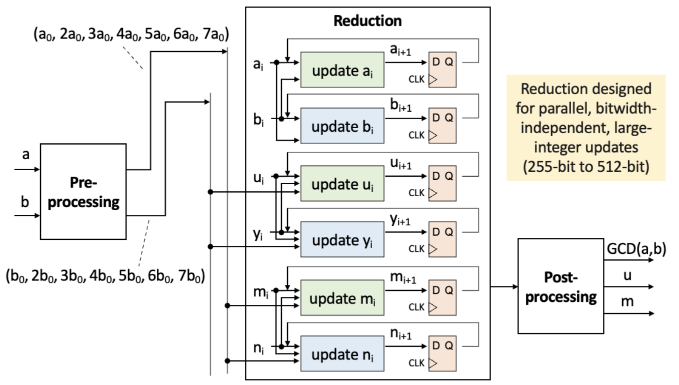


Fig. 1. Overview of dataflow in our XGCD accelerator, with variables described in Section II. Logic inside the "update u_i" unit is shown in Figure 3.

II. Dataflow Background

The full XGCD algorithm is described in [6] and consists of three stages: a pre-processing stage, a reduction stage, and a post-processing stage (Figure 1). The pre-processing stage requires carry-propagation adders (CPAs) to convert the inputs (a, b) into odd inputs to the reduction stage (a_0, b_0) such that $GCD(a, b) = GCD(a_0, b_0)$, and to precompute constant multiples of a_0 and b_0, up to 7×, in order to accelerate the reduction stage. The pre-processing stage is thus given four clock cycles to complete the CPA calculations.

The reduction stage reduces a_0, b_0 to find $GCD(a_0, b_0)$: in every iteration i, the algorithm applies a GCD-preserving transformation to reduce a_i or b_i. When a_i or b_i is even, it is divided by a power of two to generate an odd result. The largest reduction is by eight, which is allowed to leave an even result. When a_i and b_i are odd, either their sum or difference will be divisible by four, and this value is computed and divided by four. The reduction stage is guaranteed to reduce the magnitude of one operand by at least 2×, with many steps decreasing it by 4× or 8×. The iteration ends when either a_i or b_i is zero, yielding the other as the GCD.

Every cycle, the reduction stage also updates the Bézout coefficients to compute the XGCD. Since we do not know whether a_i or b_i will become zero, we maintain two equations:

Branch	a_{i+1}	b_{i+1}	δ_{i+1}	δ_{i+1} reduced like a_i,b_i?
Initialization (a or b can be even)	if odd: a_0=a else: a_0=a+b	if odd: b_0=b else: b_0=a+b	0	Not always; a_0 and b_0 may not have the same bit length
if a_i % 8 == 0	a_i >> 3	N/A	$\delta_i - 3$	Yes; δ_i and a_i reduced by the same number of bits
elif a_i % 4 == 0	a_i >> 2		$\delta_i - 2$	
elif a_i % 2 == 0	a_i >> 1		$\delta_i - 1$	
elif b_i % 8 == 0	N/A	b_i >> 3	$\delta_i + 3$	Yes; δ_i and b_i reduced by the same number of bits
elif b_i % 4 == 0		b_i >> 2	$\delta_i + 2$	
elif b_i % 2 == 0		b_i >> 1	$\delta_i + 1$	
elif δ_i[MSB] ≥ 0 (approximating $\|a_i\| \geq \|b_i\|$)	$(a_i + b_i) / 4$	N/A	$\delta_i - 1$	Not always; a_i reduced by 2 if $(a_i + b_i)$ has no carry out
	$(a_i - b_i) / 4$			Not always; we can reduce > 1 bit with $(a_i - b_i)$
elif δ_i[MSB] < 0 (approximating $\|b_i\| > \|a_i\|$)	N/A	$(b_i + a_i) / 4$	$\delta_i + 1$	Not always; b_i reduced by 2 if $(a_i + b_i)$ has no carry out
		$(b_i - a_i) / 4$		Not always; we can reduce > 1 bit with $(b_i - a_i)$

Fig. 2. High-level control and updates for a_i, b_i, and δ each cycle.

$u_i * a_0 + m_i * b_0 = a_i$ and $y_i * a_0 + n_i * b_0 = b_i$. Only one set of these variables is updated each cycle: u_i, m_i when a_i is reduced and y_i, n_i when b_i is reduced. All variables are stored in CS form during the reduction stage.

At the end, we compute $(u_i + y_i, m_i + n_i)$ as the Bézout coefficients, and $a_i + b_i$ as the GCD (since a_i or b_i will be zero and the other will be the GCD). We add these values since adding in CS form is much faster than a full-width zero detection to find which of a_i or b_i is zero, followed by a selection. The post-processing stage has a large CSA tree to compute these sums before the CPA and is also given four clock cycles to complete.

III. Large-Integer Approximations

When a_i, b_i are odd, we want to update the larger of a_i, b_i so that their difference remains positive, keeping a_{i+1}, b_{i+1} positive. This choice ensures the fastest convergence. Since fast magnitude comparison is not possible in CS form, we instead compute $\delta \approx \log_2(a_i) - \log_2(b_i)$ so sign(δ) can approximate sign$(a_i - b_i)$ [10]. We update δ each cycle by conservatively updating the bit length difference between a_i and b_i by the minimum number of bits reduced that cycle, which is an approximation of the true value of δ. Figure 2 shows the possible a_i, b_i conditions, along with how a_i, b_i, δ are updated in these conditions and in which conditions the δ approximation may not be accurate.

We find that 25% of the time, sign$(\delta) \neq$ sign$(a_i - b_i)$. These values can differ when a_i and b_i are odd and there is more cancellation than the minimum guaranteed cancellation. In those cases, the "incorrect" variable is updated. The algorithm is still functional since we applied a valid GCD-preserving transformation, but a_i, b_i can become negative, the outputs can have the wrong sign, and the algorithm can take longer to converge. Since our hardware handles negative numbers, the algorithm will still converge: δ will eventually switch sign to update the larger variable. We find that using δ rather than a slow full comparison increases the average cycle count by less than 15% across 128-bit to 2048-bit inputs, while reducing the critical path delay by 4×. For 512-bit inputs, updating δ every cycle requires a 10-bit carry-propagation adder, which fits within our cycle time.

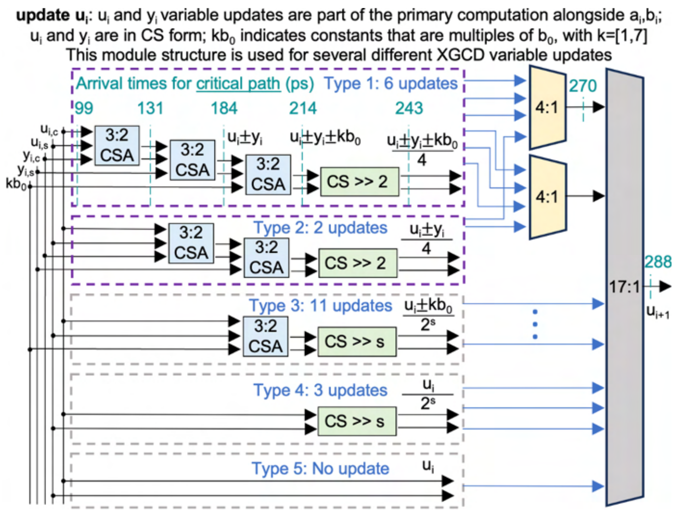

update u_i: u_i and y_i variable updates are part of the primary computation alongside a_i,b_i;
u_i and y_i are in CS form; kb_0 indicates constants that are multiples of b_0, with k=[1,7]
This module structure is used for several different XGCD variable updates
Arrival times for critical path (ps)
Type 1: 6 updates
99
131
184
214
243
270
288
u_i,c
u_i,s
y_i,c
y_i,s
kb_0
3:2 CSA
u_i±y_i
u_i±y_i±kb_0
(u_i±y_i±kb_0)/4
CS >> 2
4:1
Type 2: 2 updates
(u_i±y_i)/4
Type 3: 11 updates
(u_i±kb_0)/2^s
CS >> s
Type 4: 3 updates
u_i/2^s
Type 5: No update
u_i
17:1
u_i+1

Fig. 3. Dataflow overview for "update u_i" and arrival times on the critical path. Each dashed-outline box represents a different complexity update (there are 5 types) and indicates the number of update options of that type. "CS $>>$ 2" and "CS $>>$ s" indicate shifts in CS form, which are more complicated than normal shifts; the logic for "CS $>>$ 2" is shown in Figure 6.

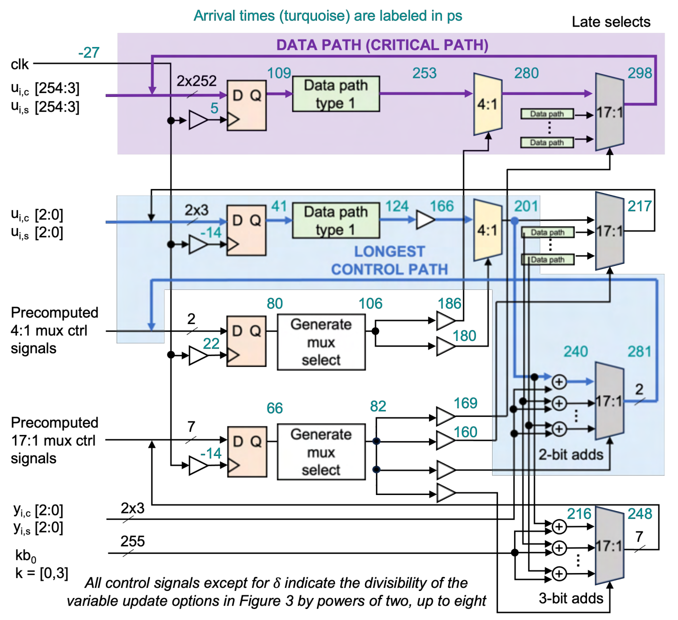


Fig. 4. Arrival times with late selects and separate computation for the LSBs for the 255-bit XGCD unit. Only the critical data path, which is of "Type 1" from Figure 3, is shown; the other data paths from Figure 3 are shown in thin green boxes. The clock arrival time is labeled such that the clock on the critical path arrives 5 ps late, where 5 ps is the setup time of the flop, making the 298 ps at the end of the data path (in purple) the cycle time. The resulting control path (in blue) traverses the LSB datapath and then goes through a 17:1 mux to be registered before driving the control for the next cycle.

IV. Control Optimizations

Since the next operation depends on the divisibility of the current results, the control delay can greatly increase cycle time: just buffering and driving these signals to the 512-bit datapaths takes ∼150 ps, which is roughly half of our final cycle time. We decrease this overhead by 77% by computing all possible updates in parallel and using late selects. This increases the overall area by 1.76×. If we selected early, we

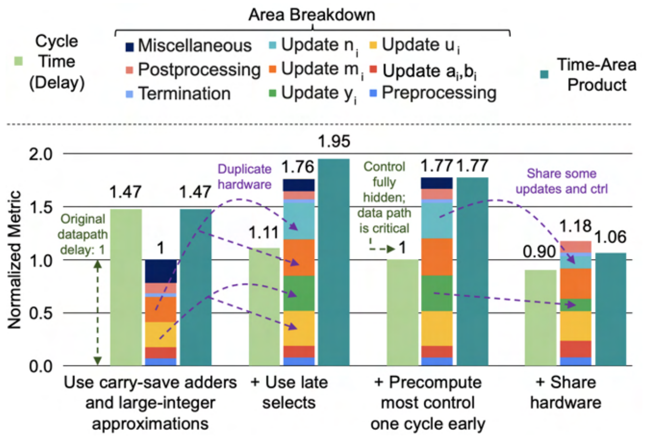


Fig. 5. Area, time, and time-area product with control logic optimizations; delay is measured relative to the datapath-limited delay.

could remove the blue blocks in Figure 1 and use "update a_i" to compute the update for a_i or b_i, "update u_i" for u_i or y_i, and "update m_i" for m_i or n_i since either a_i, u_i, m_i or b_i, y_i, n_i are updated each cycle. Using late selects also results in up to 23 separate update options for every XGCD variable, as shown in Figure 3. The design selects between the most critical updates with 4:1 multiplexers, which then feed into a 17:1 multiplexer optimized to minimize the 4:1 multiplexer path delay.

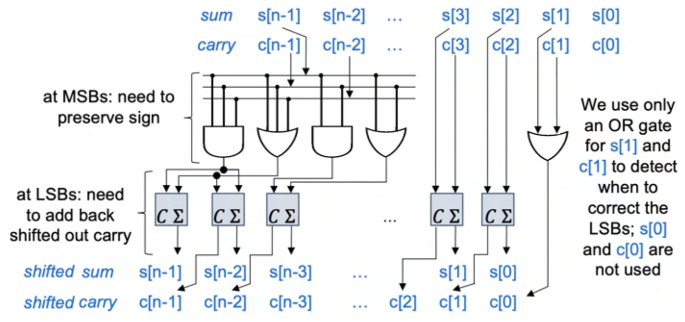
While the late selects improve timing, the control path is still critical. To hide this delay, we separate out the computation for the three least significant bits (2:0) since the control signals only depend on these bits, as shown in Figure 4. For this small amount of logic, we use more ultra-low threshold voltage and larger standard cells, which improves performance. This allows us to push the small logic to compute the control signals from the data values from the start of the next cycle to the end of the current cycle. We need to push this logic through the 17:1 multiplexer, so it is duplicated 17 times. This optimization fully hides the control path delays, marginally increases area by 0.6%, and improves timing by an additional 10%.

Finally, we realize that some different variables must have the same divisibility in this algorithm, and that some variable updates differ by only a sign. Since our hardware supports negative logic, we compute these values in only one variable update unit. Specifically, we remove the 4:1 multiplexer control generation in Figure 4, the purple dashed-outline boxes in Figure 3, and logic for the updates in the blue boxes in Figure 2 from many units. This optimization reduces the area overhead of late selects by 77% and improves timing by another 10%.

Figure 5 shows the progressive improvements of our optimizations. Compared to only using CSAs and δ, our optimizations together reduce time by 1.6$\times$ and time-area product by 1.4$\times$. The final datapath delay is 90% of the original datapath delay, since our datapath block sharing results in lower fanout and routing delays to reduce the overall datapath delay.

V. Shifting in Carry-Save Form

Shifting values in CS form requires attention at both the most significant bits (MSBs) and the least significant bits



sum
carry
s[n-1]
s[n-2]
...
s[3]
s[2]
s[1]
s[0]
c[n-1]
c[n-2]
c[3]
c[2]
c[1]
c[0]
at MSBs: need to preserve sign
at LSBs: need to add back shifted out carry
We use only an OR gate for s[1] and c[1] to detect when to correct the LSBs; s[0] and c[0] are not used
$C\Sigma$
shifted sum
shifted carry
s[n-1]
s[n-2]
s[n-3]
s[1]
s[0]
c[n-1]
c[n-2]
c[n-3]
c[2]
c[1]
c[0]

Fig. 6. Circuitry for efficiently shifting right by two in CS form ("CS >> 2" in Figure 3), which requires attention at the MSBs and at the LSBs.

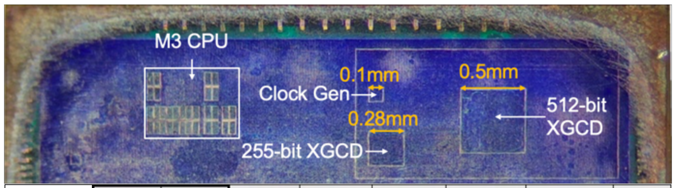
(LSBs). Since variables can become negative in our implementation, we must preserve sign when shifting in CS form. We build upon Tenca's work [11] and create relatively balanced equations for computing shifted numbers in this form. We also specialize the logic for shifting by higher powers of two to make its delay comparable to the delay for shifting by one. Our logic for shifting by two is shown in Figure 6.

If the $carry$ and sum representing a number are odd and then shifted to the right, the carry that would have been generated by adding the LSBs would have been shifted out and lost, resulting in an answer that is off by one. Consider an example with $carry = 3$ and $sum = 3$ representing a value of six. The shifted $carry$ and shifted sum would both be one, which represents a value of two (instead of three, which is six divided by two). Note that we cannot simply set the lower bit of $carry$ or sum to be one after shifting, since the LSB for the $carry$ and the sum can already be one after a shift, as in the above example. To correct the shifted result, we pass the LSBs through a half-adder to generate an empty slot at the LSB to place the carry from the dropped bits if it occurs.

We can make detecting when to apply this correction cheap: an AND gate delay in the shift-by-one case and an OR gate delay in the shift-by-two and shift-by-three cases. For the shift-by-one case, an AND gate checks whether the two lowest bits of $carry$ and sum are set. For the shift-by-two case (Figure 6), we take advantage of the fact that we already know that, if we use this result, the value mod 4 = 0, since we only shift by two when the number is divisible by four. This constraint means that the two lowest bits of $carry$ and sum have to be one of $(00, 00), (01, 11), (11, 01), (10, 10)$. All but the $(00, 00)$ case have a generated carry that would be shifted out. Thus, we can simply use an OR gate for the second lowest bits to detect when to apply this correction. If the OR gate output is one but we are not in one of these four cases, then we do not use the calculated result, so the result does not matter. Similarly, in the shift-by-three case, we detect when there is a dropped carry with an OR gate for only the third lowest bits. Our logic for shifting in CS form adds minimal overhead: Figure 3 shows that "CS >> 2" requires 29 ps on the critical path.

VI. Results

The test chip is fabricated in GlobalFoundries 12 nm FinFET technology. Our test chip was integrated on a larger SoC,



	Our 512-bit XGCD	Our 255-bit XGCD	ISSCC'23 [12]	JSSC'19 [14]	ISCAS'21 [8]	APCCAS'20 [7]	FPL'21 [9]	ePrint'20 [13]
Platform	ASIC		ASIC	ASIC / Synthesis simulation[a]	Synthesis simulation	Synthesis simulation	FPGA	Intel Coffee Lake
Technology	GF 12nm FinFET		28nm	65nm	28nm	28nm	16nm	14nm
Application	XGCD		Modular Inversion	Modular Inversion	XGCD	XGCD	XGCD	XGCD
Input Bitwidth	512	255	4096	256	1024	1024	255	255
Core Voltage (V)	0.9		0.9	1.2	—	—	—	—
Total Area (mm^2)	0.25	0.076	42.96[b]	4[b]	2.4	9.9	1847 slices; 6704 FFs	—
Average Power (mW)	565	220	4000	0.36[a]	—	—	—	—
Max Clock Frequency (GHz)	3.25	3.25	0.5	0.02	0.25	0.5	0.207	2.3
Fast vs. Constant-time Configurability	Yes	Yes	No	No	No	No	No	No
Average Execution Time (ns)	176	119[c] 87[d]	26316	36000	6490	6000	40900	2720
Constant-time Execution Time (ns)	239	119	—	36000	—	—	40900	2720

[a] Power reported for this operation from a synthesis report
[b] Total area since modular inversion area alone was not reported
[c] 255-bit XGCD measured on 255-bit XGCD unit
[d] 255-bit XGCD measured on 512-bit XGCD unit

Fig. 7. Die photo and comparison with prior work.

so Figure 7 shows only the top of the die, which contains our circuitry. We use the ARM Cortex M3 CPU from that SoC as our control processor to configure the chip and read out results. We implemented an on-chip adjustable clock generator to facilitate testing at high frequencies.

When the constant-time configuration is set, the XGCD units always run for the worst-case number of cycles for the input bitwidth. This enables constant-time execution regardless of the input values. We also have a debugging configuration mode, which allows the user to specify the number of cycles to run for before stopping, in order to enable debugging at these high GHz frequencies. After stopping execution, the unit logs all variables in CS form and all control signals. This control system runs at one-fourth the core clock frequency, enabling us to stop and inspect the computation every four cycles.

Our chip consists of an XGCD unit with 255-bit inputs and one with 512-bit inputs. We optimized the 255-bit unit for modular inversion, which often requires constant-time performance to protect against timing side-channel attacks [1], [6]. Unlike the 512-bit unit, the 255-bit unit does not reduce a_i, b_i by 4× or 8× when they are even, which eliminates many corresponding u_i, m_i, y_i, n_i update options. This makes the 255-bit unit 3.3× smaller in area, and results in 1.3× slower average performance but the same constant-time performance. Figure 7 shows that at 0.9 V, the chip computes 255-bit XGCD in 119 ns (255-bit unit) and 87 ns (512-bit unit), and 512-bit XGCD in 176 ns, when averaging execution times across random inputs. Constant-time XGCD requires 119 ns (255-bit unit) and 239 ns (512-bit unit). The 255-bit and 512-bit units occupy 0.076 mm^2 and 0.25 mm^2, respectively, and consume an average of 220 mW and 565 mW.

The bitwidth independence of our design allows us to scale our performance to the bitwidth of prior work shown in Figure 7 for comparison. For non-constant-time XGCD, our design is ∼18× faster than prior 1024-bit hardware simulations [7], [8] and 19× faster than a recent chip with a 4096-bit modular inversion unit [12]. Our chip is 8.5× to 14× smaller than [7] and 2× to 3.4× smaller than [8], after bitwidth and technology scaling. For constant-time 255-bit XGCD, our chip is 23× faster than the state-of-the-art software [13], 344× faster than prior FPGA work [9], and 303× faster than a prior chip with a 256-bit modular inversion unit [14]. Our speedup over division-based algorithms [7]–[9], [14] demonstrates the advantage of our subtraction-based XGCD algorithm for hardware, and shows that we can realize 3+ GHz XGCD core frequencies in silicon. Our chip is faster than prior subtraction-based algorithms [12], [13] due to our lower cycle counts and our control logic optimizations.

VII. Conclusion

In this paper, we present the first ASIC for the XGCD computation, which also directly computes modular inverses. By removing the need to compare carry-save values and overlapping control and data delays, our chip runs at 3.25 GHz at 0.9 V for 255-bit and 512-bit inputs, validating the advantage of our carry-save-adder subtraction-based XGCD algorithm. Our chip greatly improves XGCD performance for recent cryptographic applications, achieving a 23× speedup over software, 18× speedup over hardware simulations, and up to 303× speedup over prior chips for modular inversion.

VIII. Acknowledgements

We thank Alex Carsello, Kalhan Koul, Yuchen Mei, and Priyanka Raina for help with physical design, test board design, chip bringup infrastructure, and paper draft feedback.

References

[1] D. J. Bernstein and B.-Y. Yang, "Fast constant-time gcd computation and modular inversion," *CHES*, 2019.
[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, 1978.
[3] N. Koblitz, "Elliptic curve cryptosystems," *Math. Comput.*, 1987.
[4] B. Wesolowski, "Efficient verifiable delay functions," *Eurocrypt*, 2019.
[5] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Inf. Theory*, 1985.
[6] K. Sreedhar, M. Horowitz, and C. Torng, "A fast large-integer extended gcd algorithm and hardware design for verifiable delay functions and modular inversion," *CHES*, 2022.
[7] D. Zhu, Y. Song, J. Tian, Z. Wang, and H. Yu, "An efficient accelerator of the squaring for the verifiable delay function over a class group," *APCCAS*, 2020.
[8] D. Zhu, J. Tian, and Z. Wang, "Low-latency architecture for the parallel extended gcd algorithm of large numbers," *ISCAS*, 2021.
[9] S. Deshpande, S. M. del Pozo, V. Mateu, M. Manzano, N. Aaraj, and J. Szefer, "Modular inverse for integers using fast constant time gcd algorithm and its applications," *FPL*, 2021.
[10] R. P. Brent and H. T. Kung, "A systolic algorithm for integer gcd computation," *ARITH*, 1985.
[11] A. Tenca, S. Park, and L. Tawalbeh, "Carry-save representation is shift-unsafe: the problem and its solution," *IEEE Trans. Comput.*, 2006.
[12] G. Shi, Z. Tan, D. Cao, J. Cai, W. Zhang, Y. Wu, and K. Ma, "A 28nm 68mops 0.18 uj/op paillier homomorphic encryption processor with bit-serial sparse ciphertext computing," *ISSCC*, 2023.
[13] T. Pornin, "Optimized binary gcd for modular inversion," *ePrint*, 2020.
[14] U. Banerjee, A. Wright, C. Juvekar, M. Waller, and A. P. Chandrakasan, "An energy-efficient reconfigurable dtls cryptographic engine for securing internet-of-things applications," *JSSC*, 2019.