2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)

VLSI Architectures for Jacobi Symbol Computation

Ayan Palchaudhuri and Anindya Sundar Dhar
Department of Electronics & Electrical Communication Engineering
Indian Institute of Technology Kharagpur, Kharagpur, West Bengal, India, PIN: 721302
E-mail: ayanpalchaudhuri@gmail.com, asd@ece.iitkgp.ernet.in

***Abstract*—Jacobi symbol calculation is one of the major computation steps for certain cryptographic algorithms. In this paper, we propose a bit-sliced VLSI architecture for the same, essentially tailored for FPGA implementations that exploits the fast carry chain fabric for realizing the circuit. The architecture was essentially conceived through appropriate configuration of the target FPGA specific primitives to achieve an optimized realization in terms of delay. Our implementation outperforms behaviorally modeled circuit having similar functionality using higher levels of design abstraction, with respect to speed.**

***Index Terms*—Jacobi Symbol; FPGA; Look-Up Table; carry chain; bit-sliced design**

I. Introduction

Jacobi symbol computation is a fundamental step in many cryptographic and number theoretic applications. Examples may be cited of primality testing such as Lucas primality test [1] and Solovay Strassen primality test [2], or designing cryptosystems such as the Goldwasser Micali Cryptosystem [3] or identity based public key cryptosystem [4], [5]. Designing such applications have not only been restricted to the software domain, as VLSI implementations offer substantial hardware acceleration [1], [6].

From the perspective of VLSI, it is desirable to choose those algorithms which involve minimal or zero requirement of computing relatively complex functions such as multiplication, division or remainder operations. This is because such complex operations may significantly contribute to the critical path delay, thereby slowing down the system operation. A very poignant example may be cited of computing the greatest common divisor (GCD), where VLSI engineers often make a departure from choosing the conventional Euclid's GCD algorithm to adopting Stein's Binary GCD algorithm, as the latter replaces the conventional division operations with bit-shifts, subtraction and comparison operations [7].

Binary algorithms have been also been adopted in computing Bezout's coefficients [8] or Jacobi Symbol computation [9]. VLSI implementations for computing the Jacobi symbol have also been touched upon in [1], [6], however the treatment is done at the behavioral level using some high level hardware description language (HDL) programming constructs, without much insight into the architecture. In this paper, we propose VLSI architectures for computing the Jacobi Symbol by adopting the algorithm presented in [9]. The architectures have been conceived using the bit-sliced design paradigm, where the entire circuit is partitioned into sub-circuits, in which every sub-circuit is composed of logic cells, each configured identically, but handling a distinct pair of inputs. These logic cells are also interconnected following a regular pattern. By adopting the Huffman model for digital system design [10], we have explicitly partitioned the architecture into combinational logic and arrays of registers, as it aids in logic optimization. Though the circuit has been primarily tailored for Field Programmable Gate Array (FPGA) implementations, it is equally favourable for Application Specific Integrated Circuit (ASIC) design. Implementation on FPGA has been carried out through proper instantiation of FPGA logic primitives, as such a design philosophy generates VLSI circuits optimized for area and speed [11], [12]. The following have been our primary contributions:

- We have proposed VLSI architectures for Jacobi Symbol computation essentially tailored for FPGAs.
- A faster look-ahead based technique was also adopted to speed up the computation process.
- Our proposed architectures outperform circuits with identical functionality but realized in a fashion where the designer is relatively uninformed about the target fabric architecture for circuit deployment, and adopts a comparatively higher level of abstraction while describing the circuit through a Hardware Description Language (HDL).

The organization of the remaining paper is as follows. We revisit the Binary Jacobi algorithm and discuss the underlying principles to derive the architecture from the algorithm in Sec. II. Section III presents the design implementation results. We conclude in Sec. IV.

II. Architectures for Jacobi Symbol Computation

Xilinx Virtex-7 series of FPGAs support the state-of-the-art fabric Configurable Logic Block (CLB) architectures, where each CLB comprises of two slices. A simplified slice architecture is shown in Fig. 1. Each slice is composed of four 6-input Look-Up Tables (LUTs) primarily implementing 6-input 1-output or a 5-input dual output logic. The logic capacity of a LUT is determined by the number of inputs, and not by the complexity of the Boolean logic for those many inputs. The LUT outputs may be optionally registered using the intra-slice flip-flops (FFs). Additionally, there is a carry chain which carries out logic operations and provides dedicated routing fabric for faster signal propagation [13].

A. Binary Jacobi Algorithm

The classical Jacobi algorithm presented in [9] involves computation of remainder function which is costly both in

2380-6923/19/$31.00 ©2019 IEEE
DOI 10.1109/VLSID.2019.00076

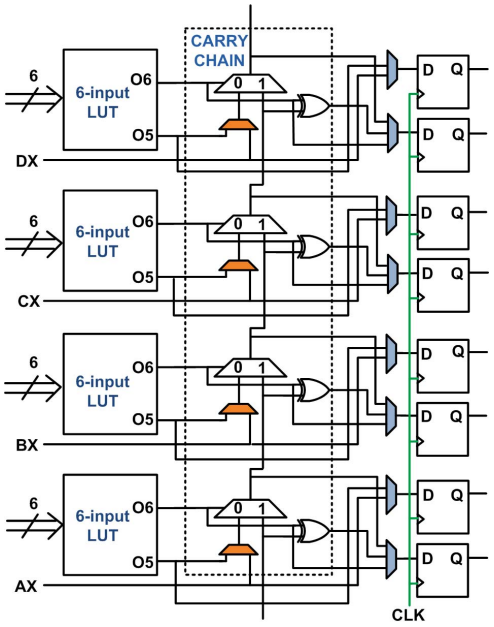

Fig. 1. A partial Virtex-7 FPGA slice logic.

Algorithm 1: Binary Jacobi Symbol Algorithm

input : Two numbers (A, B) such that $A \geqslant 0$ and $B > 0$ (B is odd)
output: Jacobi symbol $t = (\frac{A}{B})$

1 Initialize $t = 1$;
2 **while** $(A \neq 0)$ **do**
3 **while** $(A$ *is even*$)$ **do**
4 $A = \frac{A}{2}$;
5 **if** $((B \bmod 8\ (b_2b_1b_0) == 3\ (011)) \parallel$ $(B \bmod 8\ (b_2b_1b_0) == 5\ (101)))$ **then**
6 t=-t;
7 **if** $(A < B)$ **then**
8 Swap the values of A and B;
9 **if** $((A \bmod 4\ (a_1a_0) == 3\ (11))$ && $(B \bmod 4\ (b_1b_0) == 3\ (11)))$ **then**
10 t=-t;
11 $A = \frac{A-B}{2}$;
12 **if** $((B \bmod 8\ (b_2b_1b_0) == 3\ (011)) \parallel$ $(B \bmod 8\ (b_2b_1b_0) == 5\ (101)))$ **then**
13 t=-t;
14 **if** $(B == 1)$ **then**
15 Return (t)
16 **else**
17 Return (0)

terms of area and speed for hardware implementation. Instead a binary version of the algorithm was proposed in the same paper [9] that is completely bereft of such expensive operations, making it suitable for VLSI implementation. The pseudocode for the same is presented in Algorithm 1.

B. VLSI Architecture for the Jacobi Symbol Computation Unit

The overall VLSI architecture for the Jacobi symbol computation unit is shown in Fig. 2. It comprises of a datapath

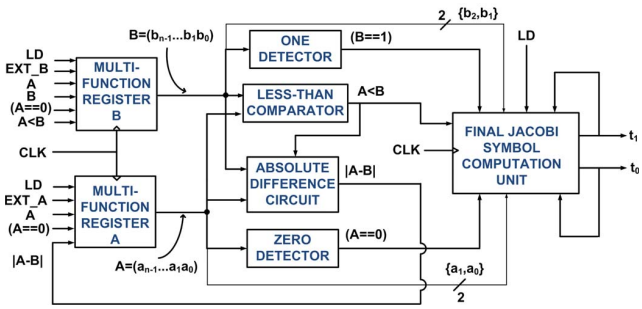

Fig. 2. Block diagram for the overall Jacobi symbol Computation Block.

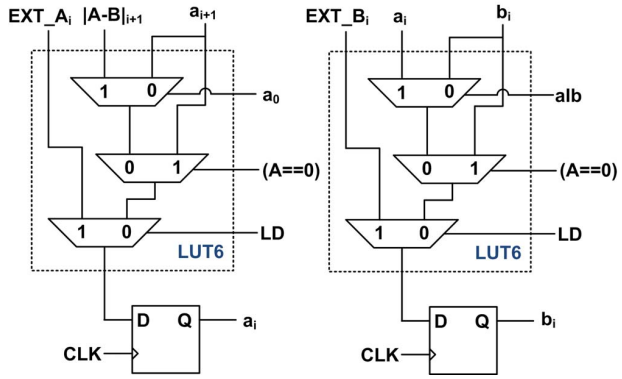

Fig. 3. Multifunction Register Configuration.

and a controlpath circuit. The datapath circuit comprises of two multifunction registers A and B, an absolute difference circuit, and a final Jacobi symbol computation unit, whereas the controlpath comprises of a zero detector, a one detector and a less-than comparator. We shall now discuss each of the circuit implementations tailored for FPGAs.

1) Multi-function Registers: The binary algorithm of Jacobi symbol computation calls for realization of 2 n-bit wide multi-function registers $A(= a_{n-1}a_{n-2}\ldots a_1a_0)$ and $B(= b_{n-1}b_{n-2}\ldots b_1b_0)$ as shown in Fig. 3. Multi-function registers are those registers which can be configured to serve multiple functional requirements on appropriate configuration of control input bits. The activity table for multi-function register A is shown in Table I. In our proposed circuit, register A gets its value updated whenever a new value is to be externally loaded into the system (EXT_A) for which we have an active high control signal LD with highest priority. Whenever the register A holds the value zero, it is an indication that the computation is over, and register A is made to retain its previous state, equivalent to *freeze* condition. A detect-0 logic $(A == 0)$ decodes this condition and serves as a control input bit to the multi-function register A. If none of the two above conditions are true, register A gets its value halved in the subsequent clock cycle whenever it is holding an even number, hence the least significant bit (LSB) of the register A itself acts as a control input to determine the same $(a_0 = 0)$. This functionality is evident from lines 3–4 of Algorithm 1, and

TABLE I
ACTIVITY TABLE FOR THE MULTI-FUNCTION REGISTER A

CONTROL / SELECT LINES			REGISTER
LD	$(A == 0)$	a_0	A
1	X	X	EXT_A
0	1	X	$A/2$
0	0	0	$A/2$
0	0	1	$(A-B)/2$

TABLE II
ACTIVITY TABLE FOR THE MULTI-FUNCTION REGISTER B

CONTROL / SELECT LINES			REGISTER
LD	$(A == 0)$	alb	B
1	X	X	EXT_B
0	1	X	B
0	0	0	B
0	0	1	A

involves a single right shift operation in hardware.

Lines 7–11 of Algorithm 1 indicate that register A needs to be conditionally updated with the contents of register B whenever $A < B$, and needs to be re-updated with a single right-shifted content of $(A - B)$ in the same iteration of the outermost while loop. The two updations have been converged to a single updation thereby saving one clock cycle per iteration in our proposed architecture. On a closer look into the algorithm, it may be observed that that the swap operation in line 7 is introduced to ensure that the $A = (A-B)/2$ operation in line 11 never turns out to be negative. A dedicated absolute difference circuit can thus compensate for the additional loss of clock cycles in the process. The architectural details of the absolute difference circuit have been taken up in Sec. II-B4. It may be noted that the combinational logic updating the contents of each bit of register A is a 6-input function and hence can be encapsulated within a single LUT.

On a similar note, register B is updated with an external data when $LD = 1$ with highest priority. The freeze operation is executed whenever the computation is over (indicated by $A == 0$) or $A \geq B$. Additionally, if $A < B$ and A is odd ($a_0 = 1$), the contents of register A is transferred to register B. The corresponding activity table for the multi-function register B is shown in Table II. Each bit-slice logic for updating register B can also be encapsulated within a single 6-input LUT, as also shown in Fig. 3. The alb control input signal is an AND-ed function of the less than comparator output detecting if $A < B$ and whether A is odd ($a_0 = 1$). A mere checking of the condition $A < B$ as depicted in Line 7 of Algorithm 1 is not sufficient form the point of view of hardware implementation. This is because every operation in software is carried out sequentially, whereas all operations are executed concurrently in hardware. Before reaching to line 7, lines 3–6 of Algorithm 1 ensures that A is no longer an even number. This check is emulated in hardware through the control signal alb. For each n-bit multi-function register, n LUTs, n FFs and $\lceil \frac{n}{4} \rceil$ slices are required.

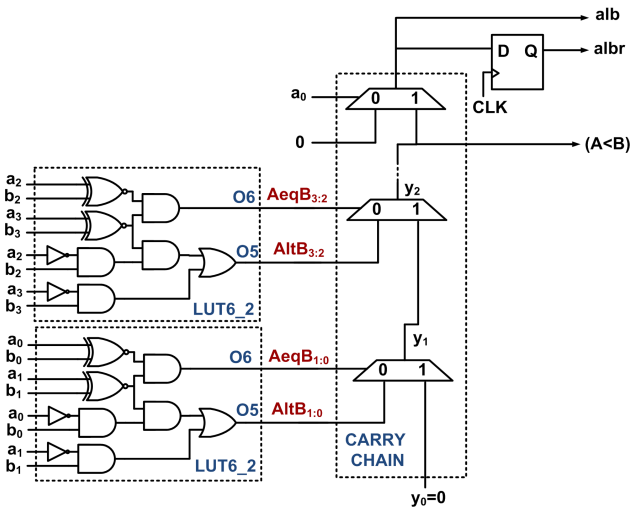

Fig. 4. Less-than Comparator.

2) Less-than Comparator: The less-than comparator configuration for FPGAs is shown in Fig. 4. Every LUT accepts two pairs of 2-bit input operands $A_{i:i-1}$ and $B_{i:i-1}$ and computes if $A_{i:i-1} = B_{i:i-1}$ through the $O6$ output of the LUT (denoted by $AeqB_{i:i-1}$) and computes if $A_{i:i-1} < B_{i:i-1}$ (denoted by $AltB_{i:i-1}$) through the $O5$ output of the LUT. The $O6$ and $O5$ responses are fed as input to the carry chain that computes $y_{\lceil \frac{i}{2} \rceil} = \overline{AeqB_{i:i-1}} AltB_{i:i-1} + AeqB_{i:i-1} y_{\lceil \frac{i}{2} \rceil - 1}$. The final output of the carry chain comparing two n-bit words is denoted by $y_{\lceil \frac{n}{2} \rceil} = (A < B)$. Thus, for comparing two n-bit words, $\lceil \frac{n}{2} \rceil$ LUTs and $\lceil \frac{n}{8} \rceil$ slices are required.

The additional check to determine if A is odd ($a_0 = 1$) and if $(A < B)$ requires AND-ing the two conditions, carried out by the carry chain realizing the original less-than comparator, instead of a LUT being deployed for the same. This is done to exploit the fast dedicated routing fabric of the carry chain instead of the relatively slow programmable routing resources of the FPGA where the carry chain output has to be transferred to the LUT input via the programmable routers. The output generated in the process alb serves as input to one of the control/select inputs for the multi-function register B. Additionally, the alb signal is also registered, shown as $albr$, which is fed as an input to the final Jacobi symbol computation unit, whose exact configuration is elaborated in Sec. II-B5.

3) Zero and One Detector: The zero and one detector circuit configurations have been shown in Fig. 5. Register A contents are checked for the value zero ($A == 0$), whereas register B contents are checked for the value one ($B == 1$). The zero detector output acts as a control signal for both multifunction registers A and B, whereas the one detector is an input to the final Jacobi symbol computation unit. The detector circuits have been realized such that each LUT accepts a 6-input sub-word $A_{i:i-5}$ or $B_{i:i-5}$, and detects if they are equal to zero, except for $B_{5:0}$ which is checked for 1. All the LUT responses are AND-ed using the carry chain fabric to yield the final response. Thus, for an n-bit zero or one detector,

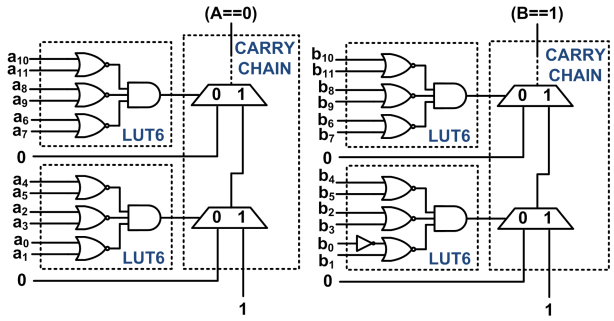

Fig. 5. Zero and One detector configuration.

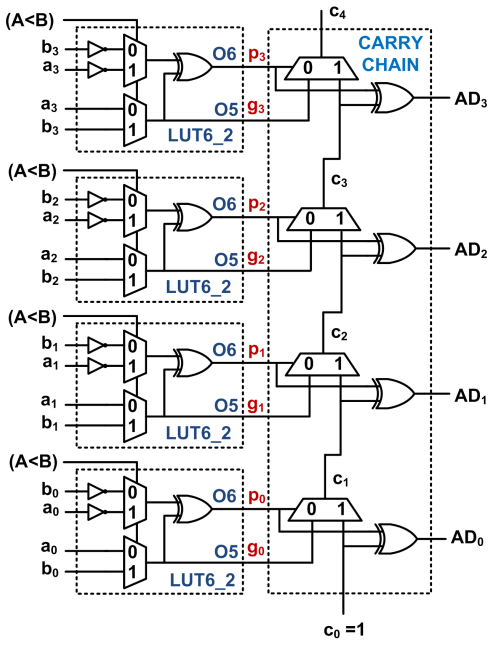

Fig. 6. Absolute difference circuit configuration.

$\lceil \frac{n}{6} \rceil$ LUTs and $\lceil \frac{n}{24} \rceil$ slices are required.

4) Absolute Difference Circuit: Circuit configuration for computing the absolute difference is shown in Fig. 6. The less-than comparator unit computing if $LT = (A < B)$ decides upon whether $(A - B = A + \overline{B} + 1)$ or $(B - A = B + \overline{A} + 1)$ should be computed using two's complement addition operation. LT thereby selects the appropriate addend and augend bit. In this implementation, we derive the *generate* bit g_i and *propagate* bit p_i as:

$$p_i = (\overline{LT}.\overline{b_i} + LT.\overline{a_i}) \oplus (\overline{LT}.a_i + LT.b_i) \quad (1)$$

$$g_i = \overline{LT}.a_i + LT.b_i \quad (2)$$

The $O6$ LUT output computes p_i, whereas the $O5$ LUT output computes g_i. It may be shown algebraically or from the truth table that $g_i = g_i\overline{p_i}$, making it feasible for a multiplexer based realization and consequently for a carry chain based implementation. Hence, the carry out c_{i+1} and

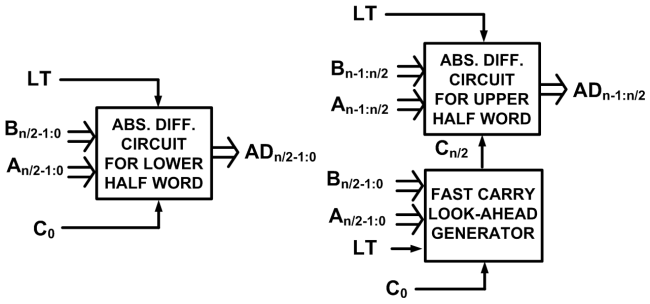

Fig. 7. Carry look-ahead scheme.

sum s_i expressions may be derived as:

$$c_{i+1} = g_i + p_i.c_i = g_i.\overline{p_i} + p_i.c_i \quad (3)$$

$$s_i = p_i \oplus c_i \quad (4)$$

Such a circuit realized for an n-bit wordlength occupies n LUTs and $\lceil \frac{n}{4} \rceil$ slices. Fast carry chain implementations for FPGAs had been previously proposed in [14]–[16]. We make an attempt to apply a similar technique for the absolute difference circuit which has a longer cascade of carry chain length in comparison to any other circuit building blocks. On closer observation, it may be seen that each configured LUT in the absolute difference circuit has two vacant inputs. The two vacant inputs may be hence utilized to accelerate the generation of the carry signal by driving them with the immediate successive pair of operands, thereby maximally utilizing the LUT structure. This carry look-ahead (CLA) based design philosophy has been expressed in Fig. 7 that computes the carry signal twice as fast compared Fig. 6. The absolute difference circuit is thereby partitioned into two halves, with each half implemented identically as shown in Fig. 6. One half accepts the lower significant inputs and the other half accepts the higher significant inputs. The circuit accepting the higher significant input bits receives its carry input from the output carry of the fast CLA generator.

From (1), we can further simplify the expression to $p_i = a_i \odot b_i$. Now, we compute the carry expression as

$$\begin{aligned} c_{i+1} &= g_i + p_i c_i \\ &= \overline{(a_i \odot b_i)}(\overline{LT}.a_i + LT.b_i) + (a_i \odot b_i)c_i \\ &= \overline{(a_i \odot b_i)}(\overline{LT}.a_i + LT.b_i) + (a_i \odot b_i) \\ &\quad (\overline{(a_{i-1} \odot b_{i-1})}(\overline{LT}.a_{i-1} + LT.b_{i-1}) \\ &\quad + (a_{i-1} \odot b_{i-1}))c_{i-1} \\ &= (LT.\overline{a_i}.b_i + \overline{LT}.a_i.\overline{b_i}) + (a_i \odot b_i)(LT.\overline{a_{i-1}}.b_{i-1} \\ &\quad + \overline{LT}.a_{i-1}.\overline{b_{i-1}}) + (a_i \odot b_i)(a_{i-1} \odot b_{i-1})c_{i-1} \\ &= ((LT.\overline{a_i}.b_i + \overline{LT}.a_i.\overline{b_i}) + (a_i \odot b_i)(LT.\overline{a_{i-1}}.b_{i-1} \\ &\quad + \overline{LT}.a_{i-1}.\overline{b_{i-1}}))\overline{(a_i \odot b_i)(a_{i-1} \odot b_{i-1})} \\ &\quad + (a_i \odot b_i)(a_{i-1} \odot b_{i-1})c_{i-1} \end{aligned} \quad (5)$$

Thus, it is evident from (5) that the logic can be mapped to a carry chain based hardware as shown in Fig. 8. We may now

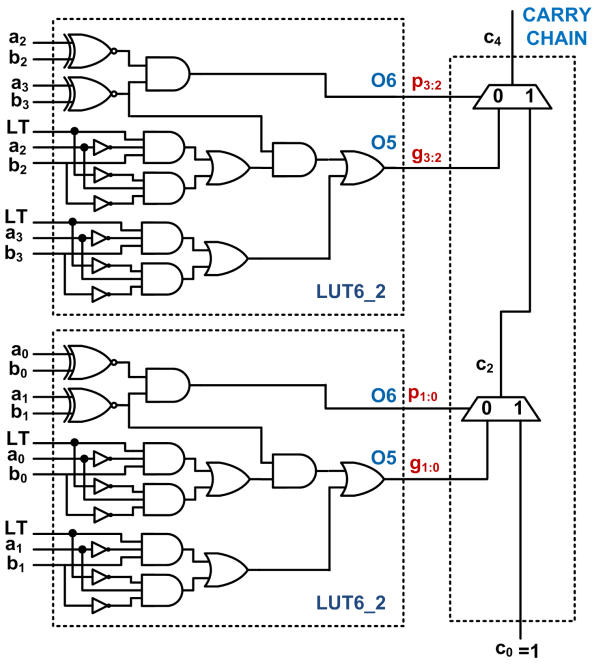

Fig. 8. Fast carry look-ahead generator.

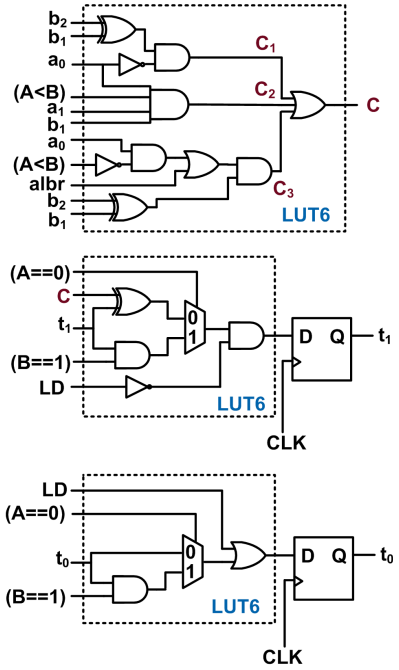

Fig. 9. Final Jacobi Symbol Computation Unit.

define a *group generate* signal $g_{i:i-1}$ and a *group propagate* signal $p_{i:i-1}$ spanning over two indices $(i : i-1)$ as

$$g_{i:i-1} = (LT.\overline{a_i}.b_i + \overline{LT}.a_i.\overline{b_i}) + (a_i \odot b_i)(LT.\overline{a_{i-1}}.b_{i-1} + \overline{LT}.a_{i-1}.\overline{b_{i-1}}) \quad (6)$$

$$p_{i:i-1} = (a_i \odot b_i)(a_{i-1} \odot b_{i-1}) \quad (7)$$

Hence the carry-out logic for the fast CLA generator governed by a single multiplexer of the carry chain may also be expressed as:

$$c_{i+1} = g_{i:i-1}\overline{p_{i:i-1}} + p_{i:i-1}c_{i-1} \quad (8)$$

The $O6$ output of the LUT computes $p_{i:i-1}$, whereas the $O5$ output of the LUT computes $g_{i:i-1}$. The fast CLA generator occupies $\lceil \frac{n}{4} \rceil$ LUTs and $\lceil \frac{n}{16} \rceil$ slices.

5) Final Jacobi Symbol Computation Unit: The final Jacobi symbol computation unit evaluates the final output stored in a 2-bit register t, initialized to 1 during loading. The register t at any instant of time may hold the value of 0, 1 or -1, encoded in the two's complement format as 00, 01 and 11 respectively. The register contents undergo a controlled two's complement operation which involves only the MSB to get complemented under three specific conditions as evident from Algorithm 1, while the LSB remains unaffected. However the entire register content may be even flushed to zero, as depicted in Line 17 of the same algorithm. We evaluate the three conditions of conditional two's complementation in the first place:

- C1: when A is even and $B \ mod \ 8 \ = \ 3$ or 5 (lines 3–6)
- C2: if $A < B$ and $A \ mod \ 4 \ = B \ mod \ 4 \ = \ 3$ (lines 7–10)
- C3: if $B \ mod \ 8 \ = \ 3$ or 5 (lines 12–13)

The control circuitry for the same is shown in Fig. 9. It may be noted that B is essentially an odd number, and also under the condition when it is loaded with the value of register A, it is guaranteed that A holds an odd integer in that clock cycle. Thus, we do not maintain any exclusive check to detect if $b_0 = 1$. This saving on the number of input bits is crucial for evaluating the three conditions C1, C2, C3 using a single LUT, and not involving LUT cascades. For condition C1, $B \ mod \ 8$ operation involves extracting the three LSBs of B. However, as B is always odd, we disregard b_0 during the check operation for detecting if $b_2b_1b_0 \ = \ 011 \ (3) \ or \ 101 \ (5)$. Hence, the check operation narrows down to detecting if $b_2 \oplus b_1 \ = \ 1$. Since A is also simultaneously checked for even data, condition $C1$ may be evaluated as $C1 = \overline{a_0}.(b_2 \oplus b_1)$. Condition C2 checks if $A \ < \ B$ and whether the last two LSBs of registers A and B are individually tied to logic 1. As $b_0 = 1$ is guaranteed, C2 may be evaluated as $C2 = (A < B).a_1.a_0.b_1$. The third condition C3 is relatively a little more involved, as the contents of register B on whose contents the $mod \ 8$ operation has to be performed, may or may not have been updated by the contents of register A indicated by lines 7–8. Here comes the utility of the signal $albr$, derived in Sec. II-B2 and shown in Fig. 4, which was the delayed version of the less-than comparator output that also checked for whether A is odd. The delayed version of this control signal is used, so as to allow register B to be updated before the $mod \ 8$ operation is performed on it. Hence, C3 may be evaluated as $C3 \ = \ ((\overline{A < B}).a_0) + albr).(b_2 \oplus b_1)$. The cumulative condition C may be obtained by the logical OR-ing of C1, C2 and C3, which will play a significant role while updating the

TABLE III
ACTIVITY TABLE FOR THE 2-BIT REGISTER $t_{1:0}$

CONTROL / SELECT INPUTS			REGISTER	
LD	$(A == 0)$	$(B == 1)$	t_1	t_0
1	X	X	0	1
0	0	X	$C \oplus t_1$	t_0
0	1	1	t_1	t_0
0	1	0	0	0

TABLE IV
IMPLEMENTATION RESULTS FOR JACOBI SYMBOL COMPUTATION UNIT

Design Style	#FF	#LUT	#Slice	Freq. (MHz)
Behavioral design	100	192	53	264.62
Prop. RC	**99**	**188**	**49**	**301.75**
Prop. CLA	**99**	**200**	**52**	**323.62**

t_1^{th} bit of the register $t_{1:0}$.

When the contents of register A become zero, register B is checked for value one in order to freeze the register t, else it is loaded with value 0, as also depicted clearly in Fig. 9. The activity table for the register t is also presented in Table III. It may also be worthwhile noting that the final Jacobi symbol computation unit has uniform complexity for any arbitrary input wordlength .

III. RESULTS AND DISCUSSIONS

The Jacobi Symbol Computation unit architecture was implemented on Xilinx Virtex-7 FPGA with XC7VX330T as the device family, package as FFG1157, and a speed grade of -2 using the Xilinx ISE 14.7 design environment. The implementation results for the proposed architectures, one with a serial ripple carry absolute difference computation circuit (denoted as Prop. RC in Table IV) and one with a CLA based absolute difference circuit (denoted as Prop. CLA) have been compared with that of a circuit designed with HDL following higher levels of design abstraction (Behavioral design). The implementation results have been presented for the circuit that accepts two 48-bit integers A and B as its inputs. The results clearly depicts that both the RC and CLA based circuits outperform the behaviorally modelled circuit in speed by 14.03% and 22.30% respectively. To the best of our knowledge, there does not exist in literature any FPGA based implementation results for the Jacobi symbol computation unit for comparison. Some implementation results exist for a finite state machine based Jacobi symbol computation unit presented in [6], however the target implementation platform is outdated, without much insight into the hardware. Moreover, there also does not exist any theoretical estimate of the complexity of VLSI architectures for the purpose of comparison.

The circuit design descriptions have been implemented on FPGAs by instantiating FPGA primitives, which gives the designer a better grip of conceiving the most optimized architecture. The Huffman style of digital design coupled with a bit-sliced design paradigm has eased the procedure of generating the circuit descriptions by running an iterative C program with a computational complexity of $O(n)$, where n is the wordlength. Under these circumstances, the method is scalable to generate the circuit descriptions for any arbitrary wordlength n. The design expressed in structural Verilog has also been synthesized using Synopsys Design Compiler with UMC 90 nm standard cell library for which the circuit occupies an area of 12350.35 μm^2, consumes a dynamic power of 2.14 mW and leakage power of 112.86 nW, and can maximally operate at a speed of 840.33 MHz.

IV. CONCLUSIONS

In this paper, we have proposed the VLSI architecture for Jacobi symbol computation using a bit-sliced design paradigm. The circuit has been primarily tailored for FPGA based implementations, however it may as well be synthesized for ASIC platforms. The architecture is well-suited for the modern day FPGAs where the 6-input LUTs and carry chains can be maximally utilized, giving rise to a compact implementation. Our proposed architectures also outperform the behaviorally modelled circuit with respect to speed for FPGA platforms.

REFERENCES

[1] A. L. Masle, W. Luk, and C. A. Moritz, "Parametrized hardware architectures for the Lucas primality test," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, July 2011, pp. 124–131.

[2] R. Solovay and V. Strassen, "A Fast Monte-Carlo Test for Primality," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 84–85, 1977.

[3] S. Goldwasser and S. Micali, "Probabilistic encryption & how to play mental poker keeping secret all partial information," in *Fourteenth Annual ACM Symposium on Theory of Computing (STOC)*, 1982, pp. 365–377.

[4] C. Cocks, "An Identity Based Encryption Scheme Based on Quadratic Residues," in *IMA International Conference on Cryptography and Coding*, 2001, pp. 360–363.

[5] F. Ţiplea and E. Simion, "New results on identity-based encryption from quadratic residuosity," Cryptology ePrint Archive, Report 2015/900, 2015, https://eprint.iacr.org/2015/900.

[6] G. Purdy, C. Purdy, and K. Vedantam, "Two Binary Algorithms for Calculating the Jacobi Symbol and a Fast Systolic Implementation in Hardware," in *49th IEEE International Midwest Symposium on Circuits and Systems (MWCAS)*, Aug 2006, pp. 428–432.

[7] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*, ser. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2010.

[8] V. Shoup, *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2005.

[9] J. Shallit and J. Sorenson, "A Binary Algorithm for the Jacobi Symbol," *ACM SIGSAM Bulletin*, vol. 27, no. 1, pp. 4–11, Jan. 1993.

[10] Z. Navabi, *Digital Design and Implementation with Field Programmable Devices*. Springer US, 2005.

[11] A. Ehliar, "Optimizing Xilinx designs through primitive instantiation," in *Proceedings of the 7th FPGAworld Conference*, 2010, pp. 20–27.

[12] A. Palchaudhuri and R. S. Chakraborty, *High Performance Integer Arithmetic Circuit Design on FPGA: Architecture, Implementation and Design Automation*. Springer India, 2016.

[13] *Xilinx Inc., "7 Series FPGAs Configurable Logic Block, User Guide, UG474 (v1.7)." [Online].*, Sep. 27 2016. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[14] A. Palchaudhuri and A. S. Dhar, "Fast Carry Chain Based Architectures for Two's Complement to CSD Recoding on FPGAs," in *14th International Symposium on Applied Reconfigurable Computing. Architectures, Tools, and Applications (ARC)*, 2018, pp. 537–550.

[15] P. Zicari and S. Perri, "A Fast Carry Chain Adder for Virtex-5 FPGAs," in *15th IEEE Mediterranean Electrotechnical Conference (MELECON)*, Apr. 2010, pp. 304–308.

[16] A. Palchaudhuri and A. S. Dhar, "High Speed FPGA Fabric Aware CSD Recoding with Run-Time Support for Fault Localization," in *31st International Conference on VLSI Design*, Jan. 2018, pp. 186–191.