Fast Hardware Computation of *x* mod *z*

J. T. Butler
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA U.S.A.

T. Sasao
Department of Computer Science and Electronics
Kyushu Institute of Technology
Iizuka, Fukuoka, JAPAN

***Abstract*—We show a high-speed hardware implementation of $x \bmod z$ that can be pipelined in $O(n-m)$ stages, where x is represented in n bits and z is represented in m bits. It is suitable for large x. We offer two versions. In the first, the value of z is fixed by the hardware. For example, using this circuit, we show a random number generator that produces more than 11 million random numbers per second on the SRC-6 reconfigurable computer. In the second, z is an independent input. This is suitable for RNS number system applications, for example. The second version can be pipelined in $O(n)$ stages.**

***Keywords:* $x \bmod z$ computation, high-speed modulo reduction, mod z arithmetic**

I. Introduction

The need for cryptographically-secure systems has inspired interest in the computation of $x \bmod z$, especially when x and z are large [6]. The $x \bmod z$ function is also useful in producing (pseudo) random numbers. Random number generators based on linear recurrences use, for example, $x_{n+1} = (P_1 x_n + P_2) \bmod N$, where x_0 is the seed value [14]. For example, Lehmer's algorithm, where the i-th random number is $s_i = as_{i-1} \mod p$, is fast enough for many simulation applications [10]. The Blum-Blum-Shub algorithm, where a bit of the random number is chosen from $s_i = s_{i-1}^2$ mod pq, such that p and q are prime, is believed to be as secure as encryption methods based on factorization [5].

Another application is the residue number system (RNS), where addition, subtraction, and multiplication are done without carries [13]. In this application, the $x \bmod z$ operation is used in the binary-to-RNS conversion, the RNS operations, and the RNS-to-binary conversion. In an RNS application, one seeks to compute simultaneously $x \bmod z$ for one value of x and several values of z. Radix converters have been proposed that use an LUT cascade [8]. Another application is in primality testing. For example, the famous polynomial-time algorithm [1] for determining if z is prime must compute $x \bmod z$. Efficient randomized primality algorithms must also compute $x \bmod z$, e.g. [2]. In this paper, we show high-speed compact hardware realizations of $x \bmod z$ suitable for implementation on an FPGA.

Surprisingly there is little work on $x \bmod z$. In one, [9] uses the sign estimate technique to estimate when the sign of $(x - qz)$ changes, where $x = (qz + x) \bmod z$. They show an algorithm for computing $x \mod z$, but no experimental results are shown. [7] discusses a unified method for computing modular multiplication, but shows no experimental results. Neither address the issue of whether an independent z can be accommodated. We consider a circuit that computes $x \bmod z$ given x and z, as n- and m-bit numbers, respectively. It can be pipelined, and each stage consists of adders and multiplexors, so the delay is small and the throughput is high. When multiple-pipelines are used, as in the case of RNS applications, the pipelines can be designed to have the same length, so that the residues of each number arrive simultaneously. We show two architectures. In the first, z is fixed by the hardware. In the second, z can be changed at each clock period. Experimental results demonstrate the efficiency of our design.

II. Basic Implementation

Our design benefits from the following viewpoint: We consider the computation of $x \bmod z$ as a *modulo reduction* process, where, at each stage, the magnitude of x is reduced, but the residue remains the same. This continues until only the residue remains.

A. Computing $x \bmod z$ for fixed z

Fig. 1 shows a circuit that realizes this process for the case where x has 8 bits and $z = 3$. First, 192 is subtracted, if possible. Next, 96 is subtracted, if possible, etc.. This circuit performs a sequence of subtract operations until only the residue OUT remains. That is, $IN = x = OUT + 3Q$, where OUT is a 2-bit remainder upon division of IN by 3 (OUT = 0, 1, or 2). Representing Q, the quotient, as an 7-bit binary number $q_6 2^6 + \ldots + q_1 2^1 + q_0 2^0$ yields

$$0 \leq IN = x = OUT + 3q_6 2^6 + \ldots + 3q_1 2^1 + 3q_0 2^0 < 256, \qquad (1)$$

where the limits $0 \leq$ and < 256 are imposed by our specification that $IN = x$ is represented as an 8-bit standard binary number.

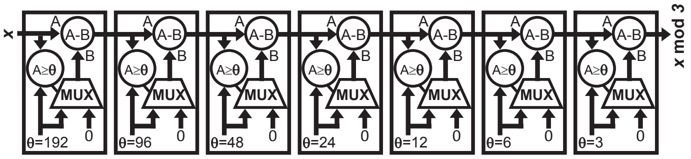

Fig. 1. Computation of $x \mod z$, where x has 8 bits and $z = 3$ (z is fixed by hardware).

The circuit shown in Fig. 1 is combinational. When n is small, such a circuit is satisfactory. However, when n is large, the delay will be too large, and it is necessary that it be pipelined. Table I shows the frequency, number of LUTs, and the number of register bits used in the Altera Stratix IV EP4SE530F43C3NES FPGA to realize $x \mod 3$, for x, an n-bit number, where n = 8, 16, 32, 64, 128, and 256, such that a pipeline register exists at the output of each stage. The resulting circuit is compact and fast. For example, for $n = 256$

TABLE I
RESOURCES ON AN ALTERA STRATIX IV EP4SE530F43C3NES FPGA NEEDED TO REALIZE $x \bmod z$, WHERE x IS AN n-BIT NUMBER AND $z = 3$ (z IS FIXED).

n	Freq.	# of r-Input LUTs				Est. # of	Total # of
	(MHz)	6-	5-	4-	3-	Packed ALMs	Registers
8	573.5	0	14	7	15	29(0%)	50(0%)
16	498.1	5	12	17	124	134(0%)	226(0%)
32	422.0	29	35	71	501	565(0%)	962(0%)
64	276.2	0	0	0	3,513	2,117(0%)	2,738(0%)
128	209.7	0	0	0	13,764	7,269(3%)	8,902(2%)
256	143.8	0	0	0	55,001	27,955(13%)	33,549(7%)

bits, only 13% of the packed ALMs available are used and the frequency exceeds 100 MHz.

Note that, in this circuit, z is fixed by the architecture. For random number generation and cryptographic applications, such a circuit is adequate. In the next section, we show a circuit where z can be changed.

B. Computing $x \bmod z$, where z is an independent variable

In the circuit in Fig. 1, the value of z is determined by the constant θ applied at each stage. Here, $\theta = 3 \cdot 2^i$, where i is an index to the stage, such that $i = 0$ corresponds to the rightmost stage and $i = 6$ corresponds to the leftmost stage. In the case of general z, $\theta = z2^i$.

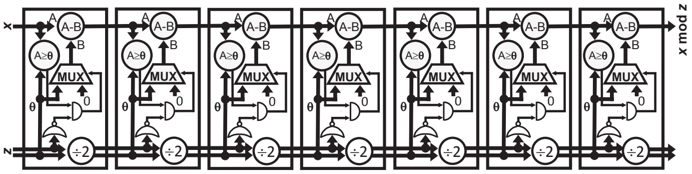

Fig. 2. Computation of x mod z (z is an independent input).

We can modify the circuit to accommodate general z, as shown in shown in Fig. 2. However, to accommodate various z, we have to

1) include enough stages to accommodate the worst case
2) allow unneeded stages to "pass-through"

For example, in the circuit in Fig. 1, which realizes $x \bmod 3$, 7 stages are needed. However, to realize $x \bmod 6$, $x \bmod 12$, ..., and $x \bmod 192$, we need 6, 5, ... , and 1 stage, respectively. Therefore, to accommodate *any* z, 7 stages are needed.

The pass-through operation can be implemented by testing the shifted value at the previous stage, as shown on the left of Fig. 2. Here, each stage has two inputs/outputs, the reduced x value (top, single bus), and a shifted version of z (bottom, double bus). The reduced x value is the same as the single rail circuit shown in Fig. 1. The shifted version of z is the replacement of the value θ_i in each stage in the circuit of Fig. 1. At the leftmost stage, z is placed on the top half of the double line showing two n-bit buses. As it passes through each stage, it is shifted down once. If there is at least one 1 bit in the upper n bits of the double bus, the NOR gate produces a 0, which, when applied to the AND gate, yields a 0 at the MUX input. This causes the MUX to deliver a 0 to the A-B gate, and the stage is a pass-through stage. Note that the two left stages are rendered unnecessary by the observation that $z > 1$. That is, for any value of $z > 1$, at least two right shifts are needed to yield all 0's in the shifted result. Therefore, we can eliminate the two leftmost stages and apply the double bus shifted twice to the next stage. On the left of Fig. 2, z, which is 3 in this example, is shifted twice right (down). In this case, the top 8 bit bus has all 0's and so the leftmost stage will *not* be a pass-through. Instead, the value of z will be tested against $192_{10} = 1100\ 0000_2$, and if it is equal to or larger than this value, 192 is subtracted from x and passed on to the stage at the right. If x is less than 192, x is passed on to the stage on the right.

Table II shows the frequency, number of LUTs, and the number of register bits used in an Altera Stratix IV EP4SE530F43C3NES FPGA to realize $x \bmod z$, where x is an n-bit number and n = 8, 16, 32, 64, 128, and 256. A pipeline register exists at the output of each stage. Note that the frequency is identical or nearly the same as with the circuit in which z is fixed (Table I). As expected, more resources are needed when z is an independent input. However, only slightly more resources are needed. That is, the price of an independent z is small.

TABLE II
RESOURCES ON AN ALTERA STRATIX IV EP4SE530F43C3NES FPGA NEEDED TO REALIZE $x \bmod z$, WHERE x IS AN n-BIT NUMBER AND z IS AN INDEPENDENT VARIABLE.

n	Freq.	# of r-Input LUTs				Est. # of	Total # of
	(MHz)	6-	5-	4-	3-	Packed ALMs	Registers
8	632.6	0	18	9	16	32(0%)	56(0%)
16	493.3	5	12	17	138	141(0%)	240(0%)
32	422.0	29	35	71	531	580(1%)	992(0%)
64	276.2	0	0	0	3,575	2,148(1%)	2,800(0%)
128	209.7	0	0	0	13,890	7,332(3%)	9,028(2%)
256	143.8	0	0	0	55,255	28,082(13%)	33,803(7%)

III. ADVANCED IMPLEMENTATION

A. Reducing complexity

These circuits have significant redundancy. For example, if x has 1,024 bits, the pipelined value of $x \bmod z$ requires 1,024 flip-flops per stage. If z is 3, then the simplest circuit shown in Fig. 1 requires 1,023 stages or a total of more than 1,000,000 flip-flops. However, not all of these are needed. For example, consider the rightmost stage in Fig. 1, which produces the output $x \bmod 3$. Only two of the eight outputs need be driven by flip-flops; all of the others can be driven by constant 0's or simply omitted. Similarly, the cell to its left need only produce three outputs driven by flip-flops. In all, 7 + 6 + 5 + 4 + 3 + 2 = 27 of the 8 × 7 = 56 stage outputs shown need to be driven by flip-flops.

Further savings can be obtained by observing that the constant term has a simple form. At the leftmost stage, this constant is $1100\ 0000_2 = 192_{10}$. It is sufficient for the comparator and subtractor to apply to three bits only.

Fig. 3 shows how these observations reduce the complexity of the middle stage in the circuit of Fig. 1. While this form of the circuit is desirable because the next step is to allow for values of z different from 3, it is useful to observe that this stage is simply realized by a 3-input 2-output function, whose truth table is shown in Table III. For example, for the first three rows, $y < 011$, and the right two bits of Stage Input

y passes to the output unchanged, as shown in the column labeled “Internal Stage $y \bmod 3$”. For the next three rows, $y \geq 011$ and $y - 011$ is passed to the output, as shown. The last two rows show don’t care values in the column labeled “Internal Stage $y \bmod 3$”. This is because the two input values $y = 110$ and 111 never occur. That is, the previous stage will reduce these values to less than $y = 110$. This can also be seen by the fact that $f_1 f_2$ in the column labeled “Internal Stage $y \bmod 3$” in this table is never 11. However, if all three inputs of the *left* stage are driven by inputs, the values $y = 110$ and 111 occur. In this case, the output $f_1 f_2$ should be 00 and 01, respectively. Because it affects one stage in the LUT cascade, the don’t care values will be chosen for *all* stages so that the left stage produces the correct output. In an FPGA, LUT’s easily realize this function. Memory-based logic is appropriate as a means to implement this circuit [16]. Fig. 4 shows the circuit of Fig. 1 as a cascade of LUTs.

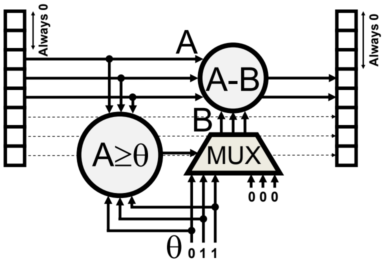

Fig. 3. Reduced complexity stage for the calculation of $x \bmod 3$.

TABLE III
TRUTH TABLE OF A SIMPLE IMPLEMENTATION OF $x \bmod z$.

Stage Input	Internal Stage $y \bmod 3$		Left Stage $y \bmod 3$	
y	f_1	f_2	f_1	f_2
000	0	0	0	0
001	0	1	0	1
010	1	0	1	0
011	0	0	0	0
100	0	1	0	1
101	1	0	1	0
110	-	-	0	0
111	-	-	0	1

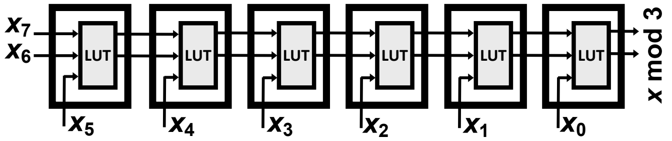

Fig. 4. Computation of $x \bmod 3$ where each stage contains an LUT.

Table IV shows the frequency and resource usage for the circuit shown in Fig. 4. It is clear that this circuit is much faster and consumes fewer resources than the two previous realizations. Like the circuit in Fig. 1, this new more compact circuit computes $x \bmod z$, where z is determined by the architecture.

B. Reducing latency

C. Tradeoff between complexity and latency

These examples show a tradeoff between the memory used to store the functions and latency. In the case of a single lookup table, a latency of one clock can be achieved, but the memory is 512 bits. One can reduce this to one-fifth, but the latency increases to eight clocks. We can make the following observation.

TABLE IV
RESOURCES ON AN ALTERA STRATIX IV EP4SE530F43C3NES FPGA NEEDED TO REALIZE $x \bmod z$, WHERE x IS AN n-BIT NUMBER AND $z = 3$ (z IS FIXED).

n	Freq. (MHz)	# of r-Input LUTs				Est. # of Packed ALMs	Total # of Registers
		6-	5-	4-	3-		
8	1520.2	0	0	0	12	14(0%)	27(0%)
16	1520.2	0	0	0	28	60(0%)	119(0%)
32	1520.2	0	0	0	60	248(1%)	495(0%)
64	1520.2	0	0	0	124	1,008(0%)	2,015(0%)
128	1097.9	0	0	0	252	1,134(0%)	2,268(0%)
256	1097.9	0	0	0	508	1,262(0%)	2,524(0%)

Lemma 1. *In the realization of $x \bmod z$ where z is fixed, let n and m be the number of bits to represent x and z, respectively. Let S be the number of stages in the pipeline, and let α be the speed-up compared to the full-latency system, where α = 1 (no speed-up), 2, 3, etc.. Let M be the total memory in bits needed in this realization. Then,*

$$S = \left\lceil \frac{n-m}{\alpha} \right\rceil, \text{ and } M = m2^{m+\alpha} \left\lceil \frac{n-m}{\alpha} \right\rceil. \qquad (2)$$

Proof: Each stage has $m + \alpha$ inputs and m outputs. Collectively, the stages have $S(m + \alpha)$ inputs, of which $(S - 1)m$ are driven by outputs from the stages. Thus,

$$S(m + \alpha) - (S - 1)m \geq n \text{ and } S = \left\lceil \frac{n-m}{\alpha} \right\rceil.$$

The observation that each unit requires $m2^{m+\alpha}$ bits, yields (2).

∎

Lemma 1 is similar to Lemma 5.1.5 of [16]. It follows from (2) that the smallest α (= 1) yields the smallest storage M. If $n - m$ is even, then $\alpha = 2$ yields the same M, since the $2^{m+\alpha}$ term doubles, while the $\lceil \frac{n-m}{\alpha} \rceil$ term halves. This can also be concluded from Theorem 9.8.2 of [16], which applies to general cascade circuits. However, as α increases beyond 2, the $2^{m+\alpha}$ dominates, and M increases rapidly. If $n - m$ is odd, this observation is approximately true. From this, we can see that there is little penalty to choosing $\alpha = 2$, and, from this point on, reducing latency increases memory significantly.

We can obtain a similar lemma for the case where z is a separate input by observing that z applies to all stages, and each stage must produce an output that depends on z. This increases by m the number of inputs, so that each unit stores $m2^{2m+\alpha}$ instead of $m2^{m+\alpha}$ bits. Thus,

Lemma 2. *In the realization of $x \bmod z$ where z is an independent input, let n and m be the number of bits to represent x and z, respectively. Let S be the number of stages in the pipeline, and let α be the speed-up over the full-latency system, where α = 1 (no speed-up), 2, 3, etc.. Let M be the total memory in bits needed in this realization. Then,*

$$S = \left\lceil \frac{n-m}{\alpha} \right\rceil, \text{ and } M = m2^{2m+\alpha} \left\lceil \frac{n-m}{\alpha} \right\rceil. \qquad (3)$$

IV. Experimental Results

A Verilog program was written for the SRC-6 reconfigurable computer that computes (pseudo) random numbers using Lehmer's algorithm [10]

$$s_{i+1} = \gamma s_i \mod z, \tag{4}$$

where the values $\gamma = 16807 = 7^5$ and $z = 2^{31} - 1$ are suggested in [14]. This same expression was also implemented by `rand`, the uniform random number generator in MATLAB Version 4 [11]. (4) is a full-period generating function, where γs_i and z can be represented in 46 and 31 bits, respectively. In our circuit, $x \mod z$ is realized using the architecture in which z is fixed (Fig. 1). A total of 16 stages (16=46-31+1) are needed. The delay of each stage is less than 5 ns. Therefore, two stages can be cascaded between pipeline registers, since the delay of these two stages is less than one 100 MHz clock period (10 ns). As a result, $x \mod z$ is realized in eight clock periods. The multiplication γs_i requires an additional stage. Thus, nine clock periods are required to generate each random number. The random number generator is realized as a producer in a producer-consumer stream. In all, it takes 36,900 clocks to generate 4,096 random numbers or 9 clocks per random number plus 36 clocks for overhead. With a clock running at 100 MHz and 9 clocks per random number, this random number generator produces more than 11 million random numbers per second.

Fig. 5 shows the first 128 random numbers. Here, a black box represents 1 and a white box represents 0. The first number, s_0, the seed, is at the left. For illustrative purposes, we chose $s_0 = 1$ (the single 1 at the least significant bit position is at the bottom). The next two numbers are the binary representation of $s_1 = 16807^1$ and $s_2 = 16807^2$, both of which are unchanged by the $\mod 2^{31} - 1$ operation. The properties of this random number generator have been extensively studied, and it is denoted as the *minimal standard generator* in [14]. Even with the non-random choice of the seed $s_0 = 1$, Fig. 5 supports the statement in [14] that the minimal standard generator is "demonstrably random".

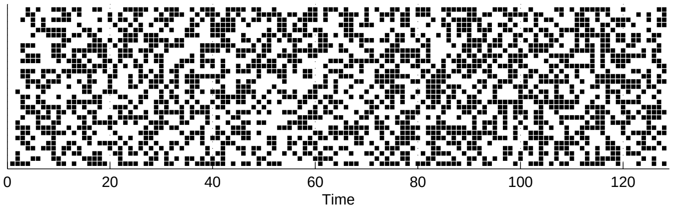

Fig. 5. Sequence of random numbers generated by the Lehmer algorithm.

Table V shows the resources used. Only a fraction of one FPGA's resources are needed. The FPGA, in this case, is the Xilinx Virtex2p XC2VP100 FPGA with Package FF1696 and Speed Grade -5. Our design met the 100 MHz timing constraint imposed by the SRC-6 Carte toolchain by a slight margin (100.4 MHz).

TABLE V
Resources on a Xilinx Virtex2p XC2VP100 FPGA used to implement the Lehmer random number generator on the SRC-6

Number of	Used/Available	Percentage
Slice Flip-Flops	2,516/88,192	2%
4-Input LUTs	2,794/88,192	3%
Occupied Slices	2,557/44,096	5%

V. Concluding Remarks

We show fast and compact circuits that realize $x \mod z$. In one version, z is fixed; its value is determined by the hardware. In another, z is an independent input. Our experimental results show that the complexity of the latter is only slightly larger than that of the former, while the speeds are nearly the same.

We illustrate the implementation of these circuits in the generation of random numbers using Lehmer's algorithm on the SRC-6 reconfigurable computer. With a clock speed of 100 MHz, we are able to produce random numbers at a rate of more than 11 million per second.

VI. Acknowledgments

This research is supported by a Knowledge Cluster Initiative (the second stage) of MEXT (Ministry of Education, Culture, Sports, Science and Technology). The authors are grateful to Dr. Jeffrey P. Hammes of SRC Computers, Inc. for help in the producer-consumer stream implementation of the Lehmer random number generator on the SRC-6. We thank four referees for comments that improved the manuscript.

References

[1] M. Agrawal, N. Kayal, and N. Saxena, "PRIMES is in P," *Annals of Mathematics*, Vol. 160, pp. 781-793, 2004.

[2] M. Agrawal and S. Biswas, "Primality and identity testing via chinese remaindering," *J. of the ACM*, Vol. 50, No. 4, pp. 429-443, July 2003.

[3] P. W. Baker, "Fast computation of A*B modulo N", *Electronic Lett.*, 16 July 1987, Vol. 23, No. 15, pp. 794-795.

[4] F. Barsi, "Mod m arithmetic in binary systems", *Inf. Proc. Lett.*, Dec. 1991, **40**, (6), pp. 303-309.

[5] L. Blum, M. Blum, and M. Shub, , "A simple unpredictable pseudo-random number generator", *SIAM Journal on Computing.*, Vol. 15, pp. 364383, May 1986.

[6] W. Diffie and M. E. Hellman, "New directions in cryptographs," *IEEE Transactions on Information Theory*, IT-22 1976 (6), pp. 644654.

[7] W. L. Freking and K. K. Parhi, "A unified method for iterative computation of modular multiplication and reduction operations," *Inter. Conf. on Comp. Des.*, pp. 80-87.

[8] Y. Iguchi, T. Sasao, and M. Matsuura,"Design methods of radix converters using arithmetic decompositions," *IEICE Trans. on Inf. and Syst.*, Vol. E90-D, No. 6, June 2007.

[9] C. K. Koc and C. Y. Hung, "Fast algorithm for modular reduction", *IEE Proc.-Comput. Digit. Tech.*, Vol. 143, No. 4, July 1998, pp. 265-900.

[10] D. H. Lehmer, "Mathematic methods in large scale computing units," *Annu. Comput. Lab. Harvard Univ.* Vol. 26, pp. 141-146, 1951.

[11] C. Moler, "Random thoughts", Cleve's Corner, MATLAB News & Notes, Fall 1995, pp. 12-13, (http://www.mathworks.com/company/newsletters/news_notes/clevescorner/).

[12] V. Ocheretnij, M. Gössel, E. S. Sogomonyan, and D. Marienfeld, "A modulo *p* checked self-checking carry select adder", *On-Line Testing Symposium, 2003, IOLTS*, 2003, pp. 25-29.

[13] A. Omondi and B. Premkumar, *Residue Number Systems: Theory and Implementation*, Imperial College Press, 2007, London, ISBN-13 978-1-86094-866-4.

[14] S. K. Park and K. W. Miller, "Random number generators: Good ones are hard to find", *Communications of the ACM*, October 1988, pp. 1192-1201.

[15] R. Sivakumar and N. J. Dimopoulos, "VLSI architectures for computing X mod m," *IEE Proc.-Circuits Devices Syst.* Vol. 142, No. 5, October 1995, pp. 313-320.

[16] T. Sasao, *Memory Based Logic Synthesis*, 1st Edition, Springer, ISBN 978-1-4419-8103-5, 2011 (to be published).