International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 1, Number 1, pages 49–62, January 2011

Efficient Exhaustive Verification of the Collatz Conjecture using DSP blocks of Xilinx FPGAs

Yasuaki Ito and Koji Nakano
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, Hiroshima, 739-8527, JAPAN

Received: July 1, 2010
Revised: October 30, 2010
Accepted: December 10, 2010
Communicated by Akihiro Fujiwara

Abstract

Consider the following operation on an arbitrary positive number: if the number is even, divide it by two, and if the number is odd, triple it and add one. The Collatz conjecture asserts that, starting from any positive number m, repeated iteration of the operations eventually produces the value 1. The main contribution of this paper is to present an efficient implementation of a coprocessor that performs the exhaustive search to verify the Collatz conjecture using a Xilinx Virtex-6 FPGA with DSP blocks, each of which contains one multiplier and one adder. The experimental results show that, our coprocessor can verify 4.99×10^8 64-bit numbers per second. Also, we have implemented a multi-coprocessors system that has 380 coprocessors on the FPGA. The experimental results show that our multi-coprocessor system can verify 1.64×10^{11} 64-bit numbers per second.

Keywords: Hardware Algorithm, Collatz conjecture, FPGA Implementation, DSP blocks, Block RAMs

1 Introduction

An FPGA (Field Programmable Gate Array) is a programmable VLSI in which a hardware designed by users can be embedded instantly. Typical FPGAs consist of an array of programmable logic blocks (or slices), memory blocks, and programmable interconnect between them. The logic block contains four- or six-input logic functions implemented by a look up table and/or several registers. Using four- or six-input logic functions, registers, and their interconnects, any combinational circuits and sequential logic can be implemented. Using design tools provided by FPGA vendors or third party companies, a hardware logic designed by users using hardware description languages can be embedded in FPGAs. Recent FPGAs except some low-end FPGAs have a DSP block with a multiplier and an adder, which can perform multiply-accumulate operation in high clock frequency [13]. It has been shown that a lot of computation can be accelerated using a circuit implemented in FPGAs [2, 3, 10, 11, 12].

The Collatz conjecture is a well-known unsolved conjecture in mathematics [4, 9, 15]. Consider the following operation on an arbitrary positive number:

even operation if the number is even, divide it by two, and

odd operation if the number is odd, triple it and add one.

The Collatz conjecture asserts that, starting from any positive number, repeated iteration of the operations eventually produces the value 1. For example, starting from 3, we have the following sequence to produce 1.

$$3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

The exhaustive verification of the Collatz conjecture is to perform the repeated operations for numbers from 1 to the infinite as follows:

for $m \leftarrow 1$ to ∞
  begin
    $n \leftarrow m$
    while $n > 1$
      if n is even **then** $n \leftarrow \frac{n}{2}$
      else $n \leftarrow 3n + 1$
  end

Clearly, if the Collatz conjecture is not true, then the while-loop in the program above never terminates for a counter example m. Working projects for the Collatz conjecture are currently checking 60-bit numbers [14] and 63-bit numbers [4].

There are several researches for accelerating the exhaustive verification of the Collatz conjecture. It is known [8] that series of even and odd operations for n can be done in one step by computing $n \leftarrow B[n_L] \cdot n_H + C[n_L]$ for appropriate tables B and C, where the concatenation of n_H and n_L corresponds to n. In [1, 5], FPGA implementations have been presented to repeat the operations of Collatz conjecture. These implementations perform the even and odd operations for some fixed size of bits of interim numbers. However, in [1], the implementation ignores the overflow. Hence, if there exists a counter example number m for the Collatz conjecture such that, infinitely large numbers are generated by the operations from m, their implementation may fail to detect it. On the other hand, in [5], the implementation can verify the conjecture for up to 23-bit numbers. This is not sufficient because working projects for the Collatz conjecture are currently checking 60-bit numbers [14] and 63-bit numbers [4].

In our previous paper [8], we have shown a software-hardware cooperative approach to verify the Collatz conjectures for 64-bit numbers m. Our approach supports almost infinitely large interim numbers n. The idea is to perform the while-loop for interim values with up to 78 bits using a coprocessor embedded in an FPGA. If an interim value n has more than 78 bits, the original value m is reported to the host PC. The host PC performs the verification for such m using unlimited number of bits by software. This software-hardware cooperative approach makes sense, because

- the hardware implementation on the FPGA is fast and low power consumption, but the number of bits for the operation is fixed,
- the software implementation on the PC is relatively slow and high power consumption, but the number of bits for the operation is unlimited.

In this paper, we improve the coprocessor architecture of our previous paper [8]. We use an embedded DSP block to further accelerate the coprocessor, while four (unsigned) 17-bit multipliers are used in our previous implementation. Let us explain more details of coprocessor architecture of our previous paper [8]. Let n be an interim number, and n_L and n_H denote the least significant 10 bits of n and the remaining bits. The coprocessor performs computation $n \leftarrow B[n_L] \cdot n_H + C[n_L]$ for the exhaustive verification of the Collatz conjecture. We have used embedded (unsigned) 17×17-bit multiplier in Xilinx Virtex-II Family FPGA XC2V3000. As illustrated in Figure 1, we have used *a parallel implementation* that includes four multipliers to compute 68×17-bit multiplication $B[n_L] \cdot n_H$. Based on this idea, we have implemented 24 coprocessors in XC2V3000 which repeatedly perform operation $n \leftarrow B[n_L] \cdot n_H + C[n_L]$ in each of 24 coprocessors in parallel. If the resulting value is overflow, that is, if n_H has more than 68 bits, the value of n is reported in the host PC. Each coprocessor can verify 2.47×10^8 64-bit numbers m per second. The remaining verification is performed for m with overflow interim value n by unbounded bit operations by software on the host PC.

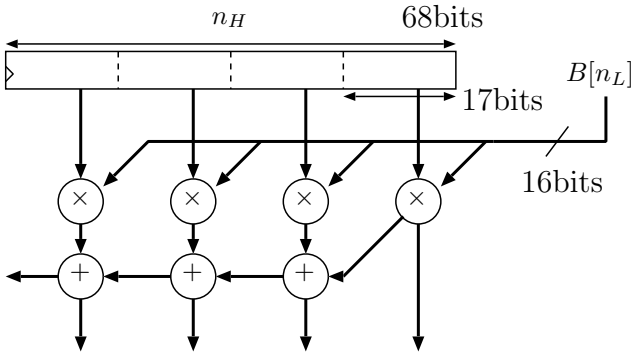

Figure 1: The parallel implementation of 68×16-bit multiplication

The main contribution of this paper is to further accelerate computation $B[n_L] \cdot n_H + C[n_L] \rightarrow n$ using a DSP block of the FPGA for each coprocessor. More specifically, computation $B[n_L] \cdot n_H$ is performed using a DSP48E1 block in Xilinx Virtex-6 Family FPGA. Figure 2 illustrates our implementation to compute $n_H \cdot B[n_L]$ using a DSP48E1 block. We have six 17-bit registers to store up to 102-bit values of n_H. The multiplication is computed in a pipeline fashion, and $17k \times 17$-bit ($k \leq 6$) multiplication can be performed in k clock cycles.

Our new implementation, *the sequential pipeline implementation* in Figure 2 has advantages over our previous implementation, the parallel implementation in Figure 1. Let us compare the sequential pipeline implementation and the parallel implementation for general case. The sequential pipeline implementation can perform $17k \times 17$-bit multiplication in k clock cycles using a single multiplier for any $k \geq 1$. On the other hand, the parallel implementation performs $17k \times 17$-bit multiplication in 1 clock cycle using k multipliers for a fixed $k \geq 1$. Suppose we have an FPGA with w multipliers and need to multiply of W pairs of $17c$-bit and 17-bit numbers. Using the sequential implementation the pair can be multiplied in c clock cycles. Thus, using w multipliers, w pairs of $17c$-bit and 17-bit numbers can be multiplied in c clock cycles. Hence, W multiplications can be performed in $\lceil \frac{W}{w} \rceil \cdot c \approx \frac{Wc}{w}$ clock cycles. The parallel implementation cannot perform the multiplication whenever $c > k$. If $c \leq k$, then a pair can be multiplied in 1 clock cycles using w multipliers. Thus, the w multipliers can multiply W pairs in $\lceil \frac{W}{\lceil \frac{w}{k} \rceil} \rceil \approx \frac{Wk}{w}$ clock cycles. Also, note that if $c < k$ then $k - c$ multipliers are not used for the multiplication. Hence, the sequential pipeline implementation is more flexible and scalable in the sense that it supports more bits and all multipliers are always used in every clock cycle. Also, since we have carefully used the sequential pipeline implementation, the clock frequency is much higher than the parallel implementation. Our sequential pipeline implementation runs in 360.490MHz, while the parallel implementation runs in 49.520MHz. Also, the overflow is detected if an interim number has more than 78 bits in our previous parallel implementation, while our new sequential pipeline implementation supports interim number up to 112 bits. Thus, the probability of the overflow, which is reported to the host PC in our new implementation, is much smaller. It follows that the PC needs to verify fewer numbers if we use the sequential pipeline implementation.

Table 1 summarizes our new implementation and previous implementation. Our parallel implementation [8] uses 4 embedded multipliers and runs in 49.520MHz. Also it can perform the verification for 2.47×10^8 numbers per second using one coprocessor. On the other hand, our new sequential pipeline implementation runs in 360.490MHz and it can verify 4.99×10^8 numbers per second using only one DSP48E1 block, which contains one multiplier. Although our new implementation can work at approximately 7.2 times higher frequency than our previous implementation, the new implementation is approximately 2.0 times faster than the previous one. This is because the new implementation needs several clock cycles to compute $n_H \cdot B[n_L] + C[n_L]$. However, the previous implementation can compute it in one clock cycle. Since we used the different implemen-

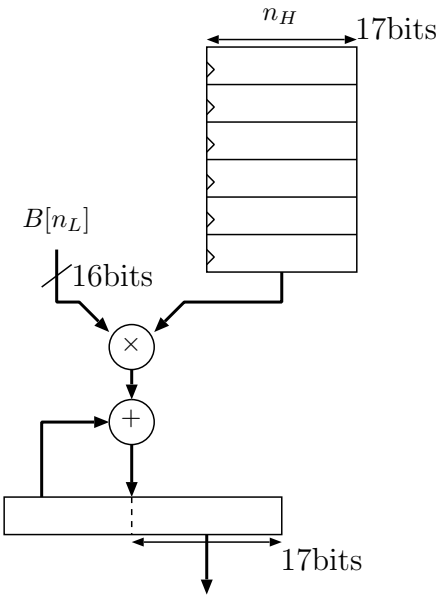

Figure 2: The sequential pipeline implementation

Table 1: Performance of coprocessors of the parallel implementation and the sequential pipeline implementation

	Parallel implementation [8]	Sequential Pipeline implementation (this paper)
Used Embedded blocks	4 multipliers	1 DSP48E1 block
Clock Frequency	49.520MHz	360.490MHz
Performance	2.47×10^8 numbers/s	4.99×10^8 numbers/s
Supported bits	78 bits	112 bits
Overflow probability	2.88×10^{-5}	1.33×10^{-15}

tation between the new implementation and the previous implementation, used technologies, types of hardware blocks, supported bit-lengths, and architectures are distinct. Therefore, the frequencies of them cannot be directly compared. However, according to the verified numbers per second that signifies the performance, we achieved a speedup factor of 2 over the previous implementation.

Also, we implemented a multi-coprocessors system that has 380 coprocessors on a Virtex-6 family FPGA XC6VLX240T-1FF1156. Since the size of the circuit is quite large, the circuit delay is increased. Then, all coprocessors run in 311.218MHz and each coprocessor can verify 4.31×10^8 64-bit numbers per second. Since each coprocessor can work independently, our implementation can verify 1.64×10^{11} 64-bit numbers per second.

In our proposed method, we implemented a coprocessor with DSP blocks. Namely, we utilize programmable logic blocks as flexible components and DSP blocks as high performance elements. From the point of view of parallelization, we use a combination of flexibility of the fine-grained structure and high performance of the coarse-grained elements. Also, we implement a multi-coprocessors system that has 380 coprocessors. In other words, we use the coprocessors as more coarse-gained elements.

This paper is organized as follows: Section 2 presents a hardware algorithm for accelerating the verification of the Collatz conjecture. Section 3 shows how the block RAM and the DSP block is used to implement the hardware algorithm. In Section 4, we show an evaluation of overflow in the verification. In Section 5, we evaluate the performance of our sequential pipeline implementation. Section 6 presents a multi-coprocessors system and its performance. Finally, Section 7 is a brief conclusion.

2 Accelerating the verification of the Collatz conjecture

The main purpose of this section is to introduce a hardware algorithm for accelerating the verification of the Collatz conjecture. The basic ideas of acceleration are shown in [9, 15].

The first technique is to terminate the operations before the iteration produces 1. Suppose that we have already verified that the Collatz conjecture is true for numbers less than n, and we are now in position to verify it for number n. Clearly, if we repeatedly execute the operations for n until the value is 1, then we can confirm that the conjecture is true for n. Instead, if the value becomes n' for some n' less than n, then we can guarantee that the conjecture is true for n because it has been proved to be true for n'. Thus, it is not necessary to repeat this operation until the value is 1, and we can terminate the iteration when, for the first time, the value is less than n.

The second technique is to perform several operations in one step. Consider that we want to perform the operations for n and let n_L and n_H be the least significant two bits and the remaining bits of n. In other words, $n = 4n_H + n_L$ holds. Clearly, the value of n_L is either 00, 01, 10, or 11. We can perform the several operations for n based on n_L as follows:

$n_L = 00$**:** Since two even operations are applied, the resulting number is n_H.

$n_L = 01$**:** First, odd operation is applied and the resulting number is $(4n_H + 1) \cdot 3 + 1 = 12n_H + 4$. After that, two even operations are applied, and we have $3n_H + 1$.

$n_L = 10$**:** First, even operation is performed and we have $2n_H + 1$. Second, odd operation is applied and we have $(2n_H + 1) \cdot 3 + 1 = 6n_H + 4$. Finally, by even operation, the value is $3n_H + 2$.

$n_L = 11$**:** First, odd operation is applied and we have $(4n_H + 3) \cdot 3 + 1 = 12n_H + 10$. Second, by even operation, the value is $6n_H + 5$. Again, odd operation is performed and we have $(6n_H + 5) \cdot 3 + 1 = 18n_H + 16$. Finally, by even operation, we have $9n_H + 8$.

For example, if $n_L = 11$ then we can obtain $9n_H + 8$ by applying 4 operations, odd, even, odd, and even operations in turn. Let B and C be tables as follows:

	B	C
00	1	0
01	3	1
10	3	2
11	9	8

Using these tables, we can perform the following table operation, which emulates several odd and even operations:

table operation For least significant two bits n_L and the remaining most significant bits n_H of the value, the new value is $B[n_L] \cdot n_H + C[n_L]$.

Let us extend the table operation for least significant two bits to d bits. For an integer $n \geq 2^d$, let n_L and n_H be the least significant d bits, that is, $n = 2^d n_H + n_L$. We call d is *the base bits*. Suppose that, the even or odd operations are repeatedly performed on $n = 2^d n_H + n_L$. We use two integers b and c such that $n = b \cdot n_H + c$ to denote the current value of n. Initially, $b = 2^d$ and $c = n_L$. We repeatedly perform the following rules for b and c.

even rule If both b and c are even, then divide them by two.

odd rule If c is odd, then triple b, and triple c and add one.

These two rules are applied until no more rules can be applied, that is, until b is odd and c is even. It should be clear that, even and odd rules correspond to even and odd operations of the Collatz conjecture. If i even rules and j odd rules applied, then the value of b is $2^{d-i}3^j$. Thus, exactly d even rules are applied until the termination. After the termination, we can determine the value of elements in tables B and C such that $B[n_L] = b$ and $C[n_L] = c$. Using tables B and C, we can perform the table operation for d bits n_L, which involves d even operations and zero or more odd operations. In this way, we can accelerate the operation of the Collatz conjecture. In our previous paper [1], we have implemented for various numbers of bits of n_L. Our implementation results show that the performance is well balanced when the number of bits of n_L is 10.

The third technique to accelerate the verification of the Collatz conjecture is to skip numbers n such that we can guarantee that the resulting number is less than n after the table operation. For example, suppose we are using two bit table and $n_H > 0$. If $n_L = 00$ then the resulting value is n_H, which is less than n. Thus, we can skip the table operation for n if $n_L = 00$. If $n_L = 01$ then the resulting value is $3n_H + 1$, which is always less than $n = 4n_H + 1$, and we can skip the table operation. Similarly, if $n_L = 10$ then we can skip the table operation. On the other hand $n_L = 11$ then the resulting value is $9n_H + 8$, which is always larger than n. Therefore, the Collatz conjecture is guaranteed to be true whenever $n_L \neq 11$, because it has been verified true for numbers less than n. Consequently, we need to execute the table operation for number n such that $n_L = 11$.

We can extend this idea for general case. For least significant d bits n_L, we say that n_L is not *mandatory* if the value of b is less than 2^d at some moment while even and odd rules are repeatedly applied. We can skip the verification for non mandatory n_L. The reason is as follows: Consider that for number n, we are applying even and odd rules. Initially, $b = 2^d$ and $c \leq 2^d - 1$ hold. Thus, while even and odd rules are applied, $b > c$ always hold. Suppose that $b \leq 2^d - 1$ holds at some moment while the rules are applied. Then, the current value of n is

$$bn_H + c < bn_H + b \leq (2^d - 1)n_H + b = 2^d n_H < n.$$

It follows that, the value is less than n when the corresponding even and odd operations are applied. Therefore, we can omit the verification for numbers that have no mandatory least significant bits.

For least significant d-bit number, we use table S to store the mandatory least significant bits. Let s_d be the number of such mandatory least significant bits. Using these tables, we can write a verification algorithm as follows:

for $m_H \leftarrow 1$ **to** $+\infty$ **do**
  for $i \leftarrow 0$ **to** $s_d - 1$ **do**

Table 2: Tables B, C, and S for least significant 4 bits.

	B	C	S
0000	1	0	0111
0001	9	1	1011
0010	9	2	1111
0011	9	2	-
0100	3	1	-
0101	3	1	-
0110	9	4	-
0111	27	13	-
1000	3	2	-
1001	27	17	-
1010	3	2	-
1011	27	20	-
1100	9	8	-
1101	9	8	-
1110	27	26	-
1111	81	80	-

begin
  $m_L \leftarrow S[i]$;
  $n \leftarrow m \leftarrow 2^d m_H + m_L$;
  while($n \geq m$) **do**
    begin
      Let n_L be the least significant d bits and
        n_H be the remaining bits.
      $n \leftarrow B[n_L] \cdot n_H + C[n_L]$;
    end
end

For the benefit of readers, we show B, C, and S for 4 base bits in Table 2. From $s_4 = 3$, we have 3 mandatory least significant bits out of 16.

For the reader's benefit, Table 3 shows the necessary word size and necessary total bits for each of tables B and C for each base bit. It also shows the expected number of odd/even operations included in one step operation $n \leftarrow B[n_L] \cdot n_H + C[n_L]$. Table 4 shows the size of table S. It further shows the ratio of the mandatory numbers over all numbers. Later, we will use two 36k-bit block RAM for tables B and C, and table S, we set base bit 10 for tables B and C, and base bit 15 for table S. Thus, in our implementation, one operation $n \leftarrow B[n_L] \cdot n_H + C[n_L]$ corresponds to expected 15 odd/even operations. Also, we skip approximately 96% of non-mandatory numbers.

3 An implementation of the computation $B[n_L] \cdot n_H + C[n_L]$ using a block RAM and DSP block

This section shows how the block RAM and the DSP block is used to implement the computation $B[n_L] \cdot n_H + C[n_L]$. For the reader's benefits, we first review the function of embedded block RAM and embedded DSP block.

Most FPGAs have embedded dual-port block RAMs as memory blocks. Figure 3 illustrates a dual-port block RAM. It has two ports A and B for which read/write operations to different address can be done in the same time. For example, Xilinx Virtex-6 family FPGAs have 36k-bit block RAMs, each of which can be configured as $32k \times 1$, $16k \times 2$, $8k \times 4$, $4k \times 9$, $2k \times 18$, $1k \times 36$ memory. See [7] for the details of the Xilinx Virtex-6 block RAM.

Table 3: The size of tables B and C.

base bit	words	word size	total bits	operation
4	16	7	112	6.0
5	32	8	256	7.5
6	64	10	640	9.0
7	128	12	1536	10.5
8	256	13	3328	12.0
9	512	15	7680	13.5
†10	1k	16	16k	15.0
11	2k	18	36k	16.5
12	4k	20	80k	18.0
13	8k	21	168k	19.5
14	16k	23	368k	21.0
15	32k	24	768k	22.5
16	64k	26	1664k	24.0

† The configuration used in our implementation.

Table 4: The size of tables S. The configuration used in our implementation is underlined.

base bit	words	word size	total bits	ratio
4	3	4	12	0.1875
5	4	5	20	0.1250
6	8	6	48	0.1250
7	13	7	91	0.1016
8	19	8	152	0.0742
9	38	9	342	0.0742
10	64	10	640	0.0625
11	128	11	1408	0.0625
12	226	12	2712	0.0552
13	367	13	4771	0.0448
14	734	14	10276	0.0448
†15	1295	15	19425	0.0395
16	2113	16	33808	0.0322

† The configuration used in our implementation.

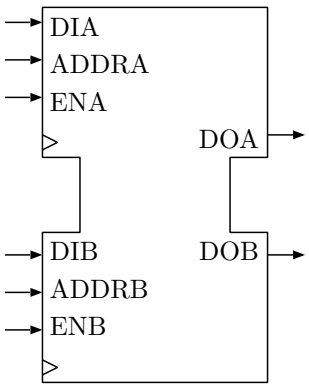

Figure 3: A dual-port block RAM

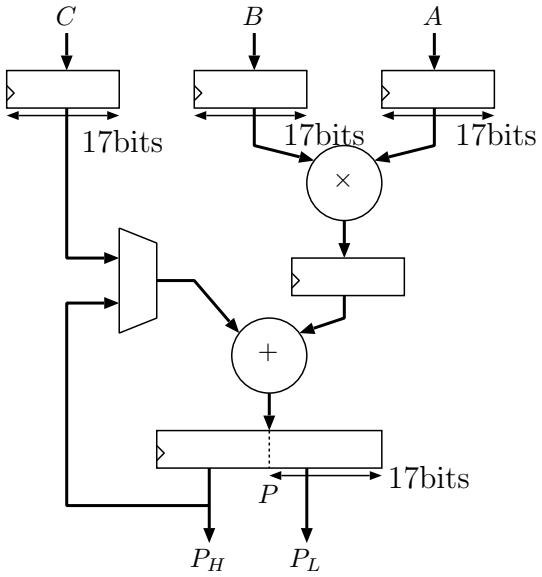

Figure 4: A part of the block diagram of Xilinx DSP48E1 block

Further, several FPGAs have embedded DSP blocks that can perform multiplication and addition in high frequency. For example, Xilinx Virtex-6 family FPGAs have DSP48E1 blocks, which can compute multiplication, addition, logic operations, overflow detection, etc. Figure 4 shows a part of the block diagram of Xilinx DSP48E1 block. One DSP48E1 block includes three input ports A, B, and C, and one output port P. In the figure, P_L and P_H denote the least significant 17 bits of P and the remaining bits. Using the DSP48E1 block, operations $P \leftarrow A \cdot B + C$ or $P \leftarrow A \cdot B + P_H$ may be performed in more than 500MHz while the same operations take less than 50MHz without using the DSP48E1 block in the FPGA. See [6] for the details of Xilinx DSP48E1 block.

In our implementation, we use 36k-bit block RAMs (Figure 3.) One block RAM is used to implement the tables B and C with base bit 10. Also, one block is used for the table S with base bit 15. As shown in Table 3, the values of B and C with base bit 10 are at most 16 bits. Thus, a 36k-bit block RAM is configured as a $1k \times 36$ memory to store the values. More specifically, for each i $(0 \leq i \leq 2^{10})$, the values of 32-bit $C[i] : B[i]$ is stored in address i of the block RAM. Using the block RAM, the values of $B[n_L]$ and $C[n_L]$ can be computed. Also, since the table S with base bit 15 needs $2k \times 15$ words from Table 4, we use one 36k-bit block RAM to store it.

A DSP block is used to compute the value of $B[n_L] \cdot n_H + C[n_L]$. As illustrated in Figure 4, a DSP block can compute $P \leftarrow A \cdot B + C$ or $P \leftarrow A \cdot B + P_H$. In the Xilinx DSP48E1 block, A and B can have up to 25 bits and 18 bits, respectively, and both C and P can have 48 bits. Also, it supports unsigned 23×17-bit multiplication and 48-bit addition. Output P is separated into 31-bit P_H and 17-bit P_L.

We implement the computation of $n \leftarrow B[n_L] \cdot n_H + C[n_L]$ as follows. The values of $B[n_L]$ and $C[n_L]$ are given to ports B and C, respectively. Let n_H is partitioned into k 17-bit unsigned integers $n_0, n_1, \ldots, n_{k-1}$ such that $n_H = n_0 \cdot 2^{0 \cdot 17} + n_1 \cdot 2^{1 \cdot 17} + \cdots + n_{k-1} \cdot 2^{(k-1) \cdot 17}$. Similarly, the resulting value p is partitioned into $k+1$ 17-bit unsigned integers $p_0, p_1, \ldots, p_k$. The values of $n_0, n_1, \ldots, n_{k-1}$ are given to port A in turn. Then, the resulting values of $p_0, p_1, \ldots, p_k$ are computed, and these values can be obtained from port P_L in turn. The key idea is to compute the value of p from the least significant digit. Register P_L is used to store the carry. The details are spelled out as follows:

1 $P \leftarrow B[n_L] \cdot n_0 + C[n_L]$;
  for $i \leftarrow 0$ to $k-2$
2 $\quad p_i \leftarrow P_L$, $P \leftarrow B[n_L] \cdot n_{i+1} + P_H$;

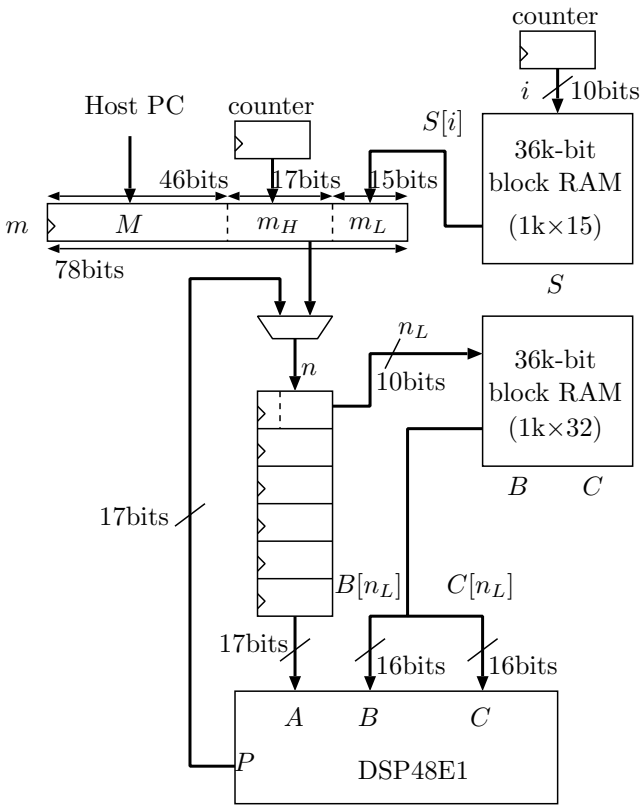

Figure 5: The architecture of a coprocessor

3 $p_{k-1} \leftarrow P_L; p_k \leftarrow P_H;$

In this hardware algorithm, each of the lines 1, 2, and 3 can be computed in one clock cycle. In this way, the values of $n_0, n_1, \ldots, n_k$ are computed.

To achieve the high clock frequency, we must use internal registers to partition the combination circuit in DSP block and perform the computation in pipeline fashion. For this purpose, we have used internal registers such that the computation is performed in three stage pipeline. Actually the number of the pipeline stages to compute $B[n_L] \cdot n_H + C[n_L]$ is four. However, the pipeline stages consist of one stage for reading $B[n_L]$ and $C[n_L]$ from the block RAM, and three stages for computing $B[n_L] \cdot n_H + C[n_L]$ with the DSP block as shown in Figure 4. Also, we have used six 17-bit registers to store p_0, p_1, ..., p_5 as illustrated in Figure 2.

Figure 5 illustrates the architecture of a coprocessor. One block RAM is used to store the values of S with base bit 15. The other block RAM is used to store the values of B and C with base bit 10. Thus, the table S has 1,295 mandatory least significant bits, a counter is used to output numbers i from 0 to 1,294. One block RAM is used to output $S[i]$. The value of m consists of a 46-bit integer M, 17-bit integer m_H, and 15-bit integer m_L. The value of M is given by the host PC, 17-bit counter is used to store the current values of m_H, and $S[i]$ determines the least significant 15 bits of m. The value of m is transferred to six 17-bit registers, which store the current value of n. The least significant 15 bits of n, n_L is given to the block RAM for tables B and C. The block RAM outputs the values of $B[n_L]$ and $C[n_L]$, which are given to ports B and C of the DSP48E1 block. After that, $B[n_L] \cdot n_H + C[n_L]$ is computed using the DSP48E1 block. If $n \geq m$ then the resulting value is stored in the register for n. Otherwise these registers are updated by next values of m.

For given 46-bit integer M by the host PC, the coprocessor performs the exhaustive verification

for the Collatz conjecture for numbers from $M \cdot 2^{32}$ to $(M+1) \cdot 2^{32} - 1$. In other words, the verification can be performed for up to 78-bit numbers. This is sufficient because working projects for the Collatz conjecture are currently checking 60-bit numbers [14] and 63-bit numbers [4].

4 The probability of overflow

Let us evaluate how often we have the overflow. Suppose that the even and the odd operations are performed for a 64-bit number n. Note that $2^{63} \leq n < 2^{64}$ holds. Let $M(n)$ denote the maximum number until n produces, for the first time, a number less than n during the operations. For example, $M(3) = 16$ holds. Suppose that we use b-bit architecture for the even and the odd operations. The verification of n is overflow if $M(n) \geq 2^b$. Let $V(b)$ be the set of such number n, that is $V(b) = \{n \mid 2^{63} \leq n < 2^{64} \text{ and } M(n) \geq 2^b\}$. Then, the ratio $R(b)$ of the overflow of 64-bit numbers on b-bit architecture is

$$R(b) \quad = \quad \frac{|V(b)|}{2^{63}}.$$

Figure 6 shows the ratio $R(b)$ of overflow for a 64-bit integer on the b-bit architecture. The ratio $R(b)$ for $b = 66, 67, \ldots, 84$ generating a 64-bit number 10^8 times at random, and seeing if $M(n) \geq 2^b$ [8]. Using the results, we expect the ratio $R(b)$ for $b > 84$. This figure shows that the ratio $R(b)$ rapidly decreases by increasing the value of b. Recall that the parameter $b = 112$ is used in our coprocessors. From the figure, we can see $R(112) = 1.33 \times 10^{-15}$, which is small enough.

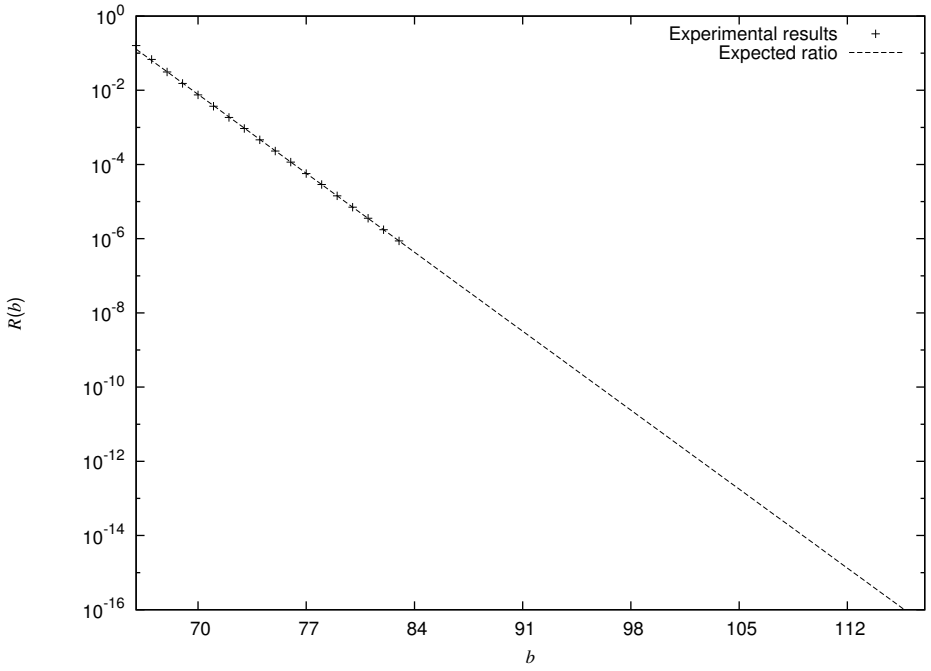

Figure 6: The ratio $R(b)$ of overflow for a 64-bit integer on the b-bit architecture

5 Experimental results

We have implemented and evaluated the performance of our sequential pipeline implementation on Xilinx Virtex-6 family FPGA (XC6VLX240T-1FF1156), which has 37,680 slices, 768 DSP48E1

blocks, and 416 36k-bit block RAMs. Table 5 shows the result of synthesis of our implementation. For the purpose of showing the goodness of our sequential pipeline implementation, the table also shows the result of synthesis of our previous implementation, the parallel implementation [8]. We have implemented and evaluated the performance of our previous implementation on Xilinx Virtex-2 family FPGA (XC2V3000-4FG676), which has 14,336 slices, 96 multipliers, and 96 18k-bit block RAMS. Since the structure of each FPGA is different, the size of each circuit is not comparable. However, compared with the number of available slices in each FPGA, each size of both implementations is small enough.

Table 5: The Results of synthesis of our implementation (single coprocessor)

	Parallel implementation [8]	Sequential pipeline implementation(this paper)
Target Device	XC2V3000-4FG676	XC6VLX240T-1FF1156
Used Embedded Block	4 multipliers	1 DSP48E1 block
Used Slices	282	103
Used RAMs	4 18k-bit block RAMs	2 36k-bit block RAMs
Maximum Clock Frequency	49.520MHz	360.490MHz

We have evaluated the computing time of the FPGA implementations by verifying the Collatz conjecture for the 64-bit numbers, because working projects for the Collatz conjecture are currently checking 60-bit numbers [14] and 63-bit numbers [4]. For this purpose, we have randomly generated 240 32-bit numbers M at random. For each generated 32-bit random number M, the FPGA implementations verified 2^{32} numbers in the range of $[M \cdot 2^{32}, (M+1) \cdot 2^{32} - 1]$. Therefore, they verified $240 \times 2^{32} \approx 1.0 \times 10^{12}$ 64-bit numbers in total. Table 6 shows the experimental results. Our new pipeline implementation is approximately 2.0 times faster than the previous implementation. Also, our previous implementation has 5,656,255 overflows in 10^{12} 64-bit numbers, while our new implementation has detected no overflow numbers. In our experiment of verifying 10^{12} 64-bit numbers, no interim number has more than 100 bits. Recall that our new implementation and previous implementation support 112-bit and 78-bit interim numbers. Thus, our previous implementation detects many overflowed interim numbers, while our new implementation has no overflow numbers. Actually, as shown in the previous section, $R(112) = 1.33 \times 10^{-15}$. Thus, we have expected 1.33×10^{-3} overflow interim numbers during the verification of 10^{12} 64-bit numbers.

Table 6: The computing time for verifying Collatz conjecture (single coprocessor)

	Parallel implementation [8]	Sequential Pipeline implementation(this paper)
Clock frequency	49.520MHz	360.490MHz
Computing time for verifying 10^{12} numbers	4,174.63 sec	2,066.35 sec
The number of verified numbers per second	2.47×10^8	4.99×10^8
The number of overflowed interim numbers	5,656,255	0

6 Multi-coprocessor system

According to the above results, we have implemented a multi-coprocessors system that has 380 coprocessors in a Virtex-6 family FPGA XC6VLX240T-1FF1156. In the system, each coprocessor is our sequential pipeline implementation as shown in the above sections. The implementation uses 37,627 slices and 380 DSP48E1, and 380 36k-bit block RAMs. Note that as shown in Table 5, each coprocessor uses two 36k-bit block RAMs for tables S, B, and C. Further, since the block RAM in the Virtex-6 FPGA is dual port, that is, it has two address ports and can be read the data of

two addresses, which can be distinct, in the same time. Hence, two coprocessors can share the two 36k-bit block RAMs for three tables. Therefore, in the multi-coprocessor system, the number of used Block RAMs is equivalent to the number of coprocessors. The timing analysis reported that our implementation runs in 311.218MHz. The frequency is lower than that of one coprocessor shown in Section 5, since the circuit delay is increased. Therefore, each coprocessor can verify 4.31×10^8 numbers per second. Because each coprocessor can work independently, our multi-coprocessor system can verify $380 \times (4.31 \times 10^8) = 1.64 \times 10^{11}$ numbers per second. Our previous approach can verify 2.89×10^9 numbers per second using 24 coprocessors [8]. As mentioned in Section 4, in our previous multi-coprocessor system, many overflowed interim numbers are detected and those numbers must be verified by software on the host PC, while our new implementation has detected no overflow numbers. Hence, the performances of them cannot be directly compared by used hardware resources and their frequencies. Therefore, to compare them, the results of synthesis and total performance are shown in Table 7.

Table 7: The Results of synthesis of our implementation (multi-coprocessor system)

	Parallel implementation [8]	Sequential pipeline implementation(this paper)
Target Device	XC2V3000-4FG676	XC6VLX240T-1FF1156
Coprocessors	24	380
Used Embedded Block	96 multipliers	380 DSP48E1 blocks
Used Slices	8,848	37,627
Used RAMs	36 18k-bit block RAMs	380 36k-bit block RAMs
Maximum Clock Frequency	40.120MHz	311.218MHz
Verified numbers per second	2.89×10^9	1.64×10^{11}

7 Conclusions

We have presented an efficient implementation of a coprocessor that performs the exhaustive search to verify the Collatz conjecture using a DSP48E1 Xilinx Virtex-6 blocks, each of which contains one multiplier and one adder.

We have implemented our coprocessor on the Virtex-6 family FPGA XC6VLX240T-1FF1156. The experimental results show that it can verify 4.99×10^8 64-bit numbers per second. Since the size of the coprocessor is small enough, several dozen coprocessors can be implemented in an FPGA. Therefore, they can work in parallel and verify the conjecture much faster.

Also, we have implemented a multi-coprocessors system that has 380 coprocessors on the FPGA. The experimental results show that our multi-coprocessor system can verify 1.64×10^{11} numbers per second.

References

[1] FengWei An and Koji Nakano. An architecture for veryfing collatz conjecture using an FPGA. In *Proc. of the International Conference on Applications and Principles of Informatin Science*, pages 375–378, 2009.

[2] J. L. Bordim, Y. Ito, and K. Nakano. Accelerating the CKY parsing using FPGAs. *IEICE Transactions on Information and Systems*, E86-D(5):803–810, May 2003.

[3] J. L. Bordim, Y. Ito, and K. Nakano. Instance-specific solutions to accelerate the CKY parsing for large context-free grammars. *International Journal on Foundations of Computer Science*, pages 403–416, 2004.

[4] Tomás Oliveira e Silva. Maximum excursion and stopping time record-holders for the $3x + 1$ problem: Computational results. *Mathematics of Computation*, 68(225):371–384, Jan 1999. Up to date computational results can be found at `http://www.ieeta.pt/~tos/3x+1.html`.

[5] Shuichi Ichikawa and Naohiro Kobayashi. Preliminary study of custom computing hardware for the 3x+1 problem. In *Proc. of IEEE TENCON 2004*, pages 387–390, 2004.

[6] Xilinx Inc. *Virtex-6 FPGA DSP48E1 Slice User Guide*, 2009.

[7] Xilinx Inc. *Virtex-6 Family Overview*, 2010.

[8] Yasuaki Ito and Koji Nakano. A hardware-software cooperative approach for the exhaustive verification of the collatz cojecture. In *Proc. of International Symposium on Parallel and Distributed Processing with Applications*, pages 63–70, 2009.

[9] Jeffrey C. Lagarias. The 3x+1 problem and its generalizations. *The American Mathematical Monthly*, 92(1):3–23, 1985.

[10] K. Nakano and E. Takamichi. An image retrieval system using FPGAs. *IEICE Transactions on Information and Systems*, E86-D(5):811–818, May 2003.

[11] K. Nakano and K. Wada. Integer summing algorithms on reconfigurable meshes. *Theoretical Computer Science*, 197:57–77, 1998.

[12] K. Nakano and Y. Yamagishi. Hardware n choose k counters with applications to the partial exhaustive search. *IEICE Trans. on Information & Systems*, 2005.

[13] Frederico Pratas, Aleksandar Ilic, Leonel Sousa, and Horácio C. Neto. Double-precision floating-point performance of computational devices: FPGAs, CPUs, and GPUs. In *Proc. of REC2010 - VI Jornadas sobre Sistemas Reconfiguráveis*, pages 83–90, 2010.

[14] Eric Roosendaal. On the 3x + 1 problem. http://www.ericr.nl/wondrous/index.html.

[15] Eric W. Weisstein. Collatz problem. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/CollatzProblem.html.