PRELIMINARY STUDY OF CUSTOM COMPUTING HARDWARE FOR THE 3X+1 PROBLEM

Shuichi Ichikawa and Naohiro Kobayashi

Department of Knowledge-based Information Engineering,
Toyohashi University of Technology
1-1 Hibarigaoka, Tempaku, Toyohashi 441-8580, Japan
E-mail: ichikawa@tutkie.tut.ac.jp

ABSTRACT

The $3x+1$ problem is a simple but unsolved problem in number theory. Though computational verifications have been attempted for this problem, they are very time-consuming. This study describes the custom hardware designs for the $3x+1$ problem, and discusses the feasibility of acceleration. A prototype hardware was implemented and evaluated with an Altera FPGA.

Keywords: Number theory, conjecture, numerical verification

1. INTRODUCTION

The $3x+1$ problem concerns the behavior of iterates of the following function:

$$T(n) = \begin{cases} (3n+1)/2, & \text{if } n \equiv 1 \pmod 2, \\ n/2, & \text{if } n \equiv 0 \pmod 2, \end{cases} \quad (1)$$

where n is a positive integer ($n = 1, 2, 3, ...$). Applying $T(n)$ recursively, the following function $T^{(k)}(n)$ is defined.

$$T^{(k)}(n) = \begin{cases} T(T^{(k-1)}(n)), & \text{if } k \geq 1, \\ n, & \text{if } k = 0. \end{cases} \quad (2)$$

The sequence $\{T^{(k)}(n) : k = 0, 1, 2, ...\}$ is called the *T-trajectory* of n. For example, the T-trajectory of 3 is $\{3, 5, 8, 4, 2, 1, 2, 1, ...\}$.

The $3x+1$ conjecture asserts that there exists some k such that $T^{(k)}(n) = 1$. In other words, the repeated application of $T(n)$ eventually produces the value 1. Although the $3x+1$ conjecture is simple to state, it is still unsolved [1]. Many studies have been done on the $3x+1$ problem and its derivatives. Some of them may be found in the references of preceding studies [1] and [2].

To find clues to solve the problem (and to find some counterexamples, if any), several computational verifications have been attempted. Oliveira e Silva [3] verified the conjecture up to 3×2^{53}, which took 4 CPU years with 4 DEC Alpha machines. Roosendaal [4] is currently extending the record with an internet-based distributed computing program.

0-7803-8560-8/04/$20.00©2004IEEE

```
for (m = start; m <= end; m += 2) {
   n = m;
   do {
      if (n & 1)  n += (n >> 1) + 1;
      else        n >>= 1;
   } while (n > m);
   /* n <= m holds here */
   if (n == m) {
      /* counterexample found! */
   }
   /* n < m : conjecture holds */
}
```

Fig. 1. An example of a 3x+1 program.

Though the $3x+1$ problem seems simple enough for hardware implementation, there have been no studies on custom circuit implementation of this problem. The present study describes the preliminary designs of custom hardware for the $3x+1$ problem with some evaluation results.

2. STRATEGIES

It is inefficient to implement the $3x+1$ problem literally. For example, we can terminate the iteration when $T^{(k)}(n) < n$, if the conjecture is already verified for $\forall m < n$. For the same reason, we do not have to examine even numbers, because an even number immediately becomes $n/2$, which is smaller than n.

Figure 1 shows an example of a $3x+1$ program. The variables $start$ and end are odd numbers, and the loop variable m is incremented by 2 to examine only odd numbers. When n is odd, $(3n+1)/2 = n + \lfloor n/2 \rfloor + 1$ is calculated. This calculation can be implemented by one shift and two additions, without using multiplication. When n is even, only one shift is required. If $n = m$ holds after the do–while statement, this m is a counterexample of the conjecture because the trajectory of m forms a loop that does not include the value 1. This do–while loop may not finish for some m, if the conjecture does not hold and the sequence of n diverges.

As mentioned above, we can skip even numbers in the $3x+1$ program. Moreover, we can skip the n such that $n \equiv 1 \pmod 4$, because $T(T(n)) < n$ holds for

all $n = 4k + 1$ $(k > 0)$ as shown below.

$$T(T(n)) = T(T(4k+1)) = T(6k+2) = 3k+1 < n.$$

This means that we only have to examine the n such that $n \equiv 3 \pmod 4$.

Another improvement is to integrate two or more steps of $T(n)$ into one. For example, assume that $n \equiv 3 \pmod 4$. We can calculate the following polynomial $T(T(n))$ instead of $T(n)$.

$$\begin{aligned} T(T(n)) &= T(T(4k+3)) = T(6k+5) \\ &= 9k + 8 = 9\lfloor n/4 \rfloor + 8 \quad (k \geq 0). \end{aligned}$$

Such methods are called *composite polynomials* [5]. The following equation chooses one of four functions according to the least two bits of n, and thus is called a 2-bit composite polynomial. This technique can be naturally generalized to k-bit composite polynomials ($k \geq 2$).

$$C(n) = \begin{cases} 3\lfloor n/4 \rfloor + 1, & \text{if } n \equiv 1 \pmod 4, \\ 3\lfloor n/4 \rfloor + 2, & \text{if } n \equiv 2 \pmod 4, \\ 9\lfloor n/4 \rfloor + 8, & \text{if } n \equiv 3 \pmod 4, \\ \text{make\_odd}(n), & \text{if } n \equiv 0 \pmod 4. \end{cases} \quad (3)$$

Here, the function make_odd(n) shifts out the lowermost zeros of n and makes $C(n)$ odd, as shown in the following piece of code:

```
make_odd(n) {
    while ((n & 1) == 0) n >>= 1;
    return n;
}
```

All these acceleration techniques (and many others) are presented by Leavens and Vermeulen [5] and Oliveira e Silva [3]. Some of their techniques are equally applicable to software and hardware, while some are specific to software implementation. Although it is very important to examine their techniques one by one, it is beyond the scope of this study. In the rest of this paper, we concentrate on the techniques described in the above discussion.

It is obvious that the variable n (in Fig. 1) has to be long enough to avoid calculation errors (overflows), but it is not evident how long it should be. The maximum excursion of the trajectory n, which is denoted by $t(n)$, is the maximum value of $T^{(k)}(n)$ for $k \geq 0$, if it exists, or infinity, otherwise. No proof is yet known, but Oliveira e Silva [3] reported that $t(n) < 8n^2$ holds for all $n < 3 \times 2^{53}$. This fact means that the variable n should be longer than $(2 \log_2 m + 3)$ bit, where m is the variable m in Fig. 1. If we are going to explore $m \sim 2^{60}$, we have to prepare 123-bit (or even longer) variables and arithmetics. Therefore, we have to use multiple-precision arithmetic for the $3x + 1$ problem.

3. CIRCUIT DESIGNS

This section describes the custom circuit designs for the $3x+1$ problem. In the present study, 140-bit arithmetic units are designed for $n < 2^{64}$ with overflow checker hardware. Even if $t(n)$ becomes exceptionally large, or even if the do–while statement in Fig. 1 diverges by any chance, such events will be detected by the overflow checker.

All designs are written in VHDL, and targeted for an Altera Stratix FPGA (EP1S10F780C7), which contains 10570 logic elements (LEs) [6]. All VHDL source codes were processed by Altera Quartus II 3.0 software for analysis, synthesis, fitting, assembly, and timing-analysis.

In this study, five design alternatives are examined. Table 1 summarizes the evaluation results of these designs for $3 \leq n \leq 2^{20}$. The logic scale and maximum operational frequency are derived from Quartus II output. The cycle count is calculated with a cycle-exact simulator of each design. The execution time is estimated from the cycle count and maximum operational frequency. Area-Time product (AT product) is the product of logic scale and execution time, and thus it is regarded as the reciprocal of *performance density*. A smaller AT product means that each unit requires a smaller area for the same performance, and that we can implement more units in the same area to squeeze more performance from there. Since the $3x + 1$ conjecture can be verified in parallel for different blocks of numbers, AT product is a very important measure for evaluation.

The first design, **Naive**, is a literal implementation of the $3x + 1$ problem. It applies $T(n)$ step by step for all n (even and odd), and thus requires many clock cycles. Although Naive operates at a relatively high clock frequency, the total execution time is rather long. The logic scale of Naive is small, but AT product is degraded by a long execution time.

The second design **CP2** implements 2-bit composite polynomials, which are shown in Eq.(3). CP2 examines only the numbers such that $n \equiv 3 \pmod 4$ to reduce cycles. The block diagram of CP2 is shown in Fig. 2. Unit A is a down-shifter to calculate $\lfloor n/4 \rfloor$ and make_odd(n). Unit B is an up-shifter to make $2\lfloor n/4 \rfloor$ and $8\lfloor n/4 \rfloor$ from $\lfloor n/4 \rfloor$. Unit C calculates the polynomials shown in Eq.(3). Unit D finds and records the maximum excursion $t(n)$. Shifters in unit A and B were implemented with `lpm_clshift` in LPM (the library of parameterized modules) [7], which provides technology-dependent library functions for Altera FPGA. The clock cycle of CP2 is only 26% of Naive. However, CP2 is twice as large in logic scale and operates at a slower clock frequency than Naive, consequently making the AT product inferior to Naive. These problems are mainly caused by unit A, which contains a 140-bit wide priority-encoder and a 140-bit wide down-shifter to implement make_odd(n).

Since the down-shift operation for make_odd can be serialized, we can shrink the logic scale of unit A in exchange for some additional clock cycles. We examined the various designs (4, 8, 16, 32, 64, 128, and 140-bit wide), and found that the 8-bit version of unit A shows the best AT product. The design **CP2/S8** is the modified version of CP2, where the unit A is 8-bit wide. The block diagram is exactly the same as CP2

Table 1. Design summary.

Design name	Logic scale (LE)	Max. freq. (MHz)	Clock cycles (cycle)	Exec. time (sec.)	AT product (LE×sec.)
Naive	1489	62.75	14104634	2.248×10^{-1}	334.7
CP2	3287	19.24	3614182	1.878×10^{-1}	617.5
CP2/S8	2081	44.11	3614256	8.194×10^{-2}	170.5
CP2/S8/A1	1941	47.13	3614256	7.669×10^{-2}	148.8
CP2/S8/A1/NMX	1758	49.19	3614256	7.348×10^{-2}	129.2

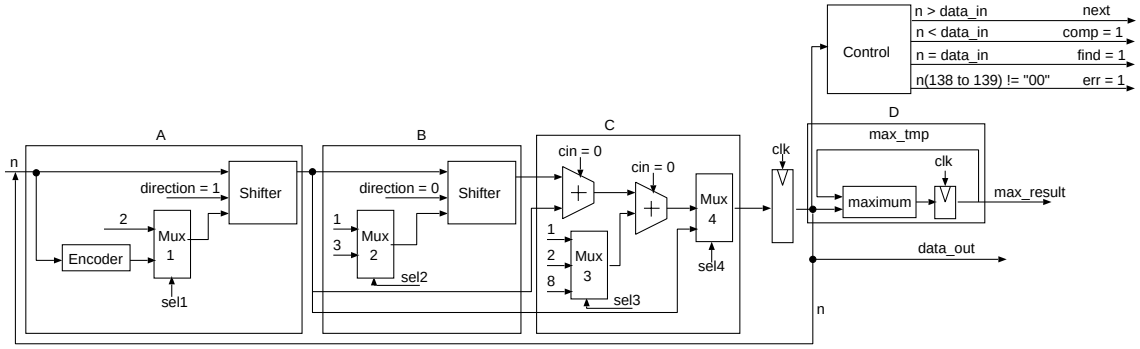

Fig. 2. Block diagram of CP2 and CP2/S8.

(Fig. 2). CP2/S8 requires 0.002% more cycles than CP2, but operates at 2.3 times higher clock frequency. CP2/S8 is much smaller and faster than CP2. It also shows a better AT product than CP2 and Naive.

Although the unit C of CP2/S8 contains two 140-bit adders, one of them is used to add a small integer (1, 2, or 8). This adder can be omitted by using the fact that the up-shifter of unit B makes the lowermost bits into zeros. The resulting design **CP2/S8/A1** contains only one adder in unit C. In this design, $9n+8$ in Eq.(3) is realized by $(8n+7)+n+1$, where $(8n+7)$ is the first operand of the adder, n is the second, and the last $+1$ is fed as carry-in. This improvement makes the circuit smaller and faster, as shown in Table 1.

There is still some room for improvement. CP2/S8/A1 includes the multiplexer MUX4 in unit C (Fig. 2). Since MUX4 is used only when n is even, the function of MUX4 can be substituted by the adder in unit C, if the adder is properly controlled. By this improvement, a 140-bit wide bus to MUX4 is also reduced. The resulting design, **CP2/S8/A1/NMX**, is illustrated in Fig. 3. CP2/S8/A1/NMX is smaller and faster than CP2/S8/A1 as shown in Table 1.

Many other designs, which adopt other LPM libraries (`lpm_mux` and `lpm_add_sub`), were examined but declined, because their AT products were worse than the designs shown in this section. Various adder configurations (e.g., carry-select adder) were also evaluated and declined.

4. EVALUATION RESULTS

The design CP2/S8/A1/NMX was implemented and verified using an Altera NIOS development board with an APEX20K FPGA [8] (EP20K200EFC484-2X). As the purpose of this implementation is the verification of the design, additional features were also embedded for debug support. The circuit was estimated to operate at 18 MHz clock, but a conservative 8.3 MHz clock (1/4 of 33.3 MHz system clock) was selected to leave some margin. The execution for $3 \leq n \leq 2^{23}$ took 3.5 seconds as expected, and the hardware cycle counter displayed exactly the same value as the cycle simulator predicted. All other debug registers showed the expected values. We checked the results carefully and concluded that the circuit is fully operational.

A performance evaluation with a Stratix FPGA is also underway. Since a Stratix can implement multiple-precision arithmetic more effectively than APEX20K, it is expected to show better results for the $3x+1$ problem.

For comparison, we measured the execution time of software implementations for $3 \leq n \leq 2^{20}$. These codes are written in C language and compiled by gcc-3.2 with optimization (-O). They are executed on an Intel Pentium 4 2.53 GHz processor with 1 GB main memory and FreeBSD 4.6.2. The first program (**Basic**) implements 128-bit variables and arithmetics, and only examines the cases $n \equiv 3 \pmod 4$. We further added 2-bit composite polynomials (**CP2**) and 4-bit composite polynomials (**CP4**) to Basic implementation. The results are listed in Table 2.

Table 1 and Table 2 show that a single unit of CP2/S8/A1/NMX of 49 MHz is faster than the software Basic+CP2 on 2.53 GHz Pentium 4. Considering that one EP1S10 FPGA can contain 5 units, an EP1S10 can be about 6 times faster than Basic+CP2. If an EP1S80 is used, the performance gain can be 56

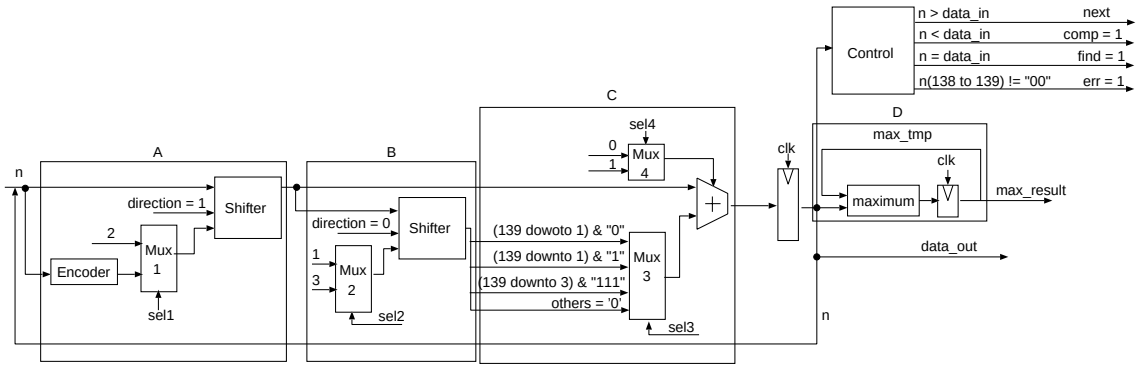

Fig. 3. Block diagram of CP2/S8/A1/NMX.

Table 2. Software execution time.

Program	Exec. Time (sec.)
Basic	1.05×10^{-1}
Basic+CP2	9.52×10^{-2}
Basic+CP4	6.66×10^{-2}

at maximum. This fact suggests that the custom circuit on FPGA can be competitive against the software on an off-the-shelf microprocessor for this problem, if the algorithm is the same.

However, Table 2 shows that Basic+CP4 is 43% faster than Basic+CP2. Apparently, the performance of the software can be improved by applying more techniques. Oliveira e Silva [3] reported that their program tests an interval of 3.17×10^8 integers each second on DEC a Alpha 266 MHz machine. This suggests that their program takes 3.3 milliseconds for $3 \leq n \leq 2^{20}$, which is 20 times faster than Basic+CP4. Considering that their program was executed on a 266 MHz Alpha processor, the performance may be 100 times or more on a 2.53 GHz Pentium 4. This may offset the performance gain of custom hardware, described in the previous paragraph.

5. CONCLUSION

After all these discussions, the authors still believe in this approach because the custom circuit can be further improved. Not all, but many techniques are applicable to hardware as well as software. If more cut-off techniques [5] are applied to custom circuit designs, hardware performance would be much improved. More extensive use of composite polynomials is also worth considering. The hardware-specific techniques (e.g., pipelining) are promising for greater performance. The use of multiple FPGA chips and multiple FPGA boards may be sometimes the simplest and the most cost-effective solution. Further investigation of these items is required, but it is beyond the scope of this preliminary study.

In number theory, there are many problems and conjectures that require numerical verification, including the derivatives of the $3x + 1$ problem. Reconfigurable systems may become powerful new equipment for these problems.

6. ACKNOWLEDGMENTS

This work was partially supported by a Grant-in-Aid for Scientific Research from the Japan Society for the Promotion of Science (JSPS). Support for this work was also provided by the 21st Century COE Program "Intelligent Human Sensing" from the Ministry of Education, Culture, Sports, Science and Technology.

7. REFERENCES

[1] R. Guy, *Unsolved Problems in Number Theory*, 2nd ed. Springer-Verlag, New York, 1994, ch. E16.

[2] J. Lagarias, "The 3x+1 problem and its generalizations," *American Math. Monthly*, vol. 92, no. 1, pp. 3–23, 1985.

[3] T. Oliveira e Silva, "Maximum excursion and stopping time record-holders for the 3x+1 problem: Computational results," *Math. Comput.*, vol. 68, no. 225, pp. 371–384, 1999.

[4] E. Roosendaal, "On the 3x+1 problem," http://personal.computrain.nl/eric/wondrous/.

[5] G. Leavens and M. Vermeulen, "3x+1 search programs," *Computers Math. Applic.*, vol. 24, no. 11, pp. 79–99, 1992.

[6] Altera Corp., *Stratix Device Handbook*, Jul. 2003, http://www.altera.com/.

[7] ——, *LPM Quick Reference Guide*, Dec. 1996, http://www.altera.com/.

[8] ——, *APEX20K Programmable Logic Device Family*, Feb. 2002, http://www.altera.com/.