AREA-EFFICIENT IMPLEMENTATION OF A FAST SQUARE ROOT ALGORITHM

Matti T. Tommiska
Laboratory of Signal Processing and Computer Technology
Helsinki University of Technology
P.O. Box 3000
FIN-02015 TKK
Finland
Phone +358 9 451 2477 Fax +358 9 460 224
Matti.Tommiska@hut.fi

ABSTRACT

In this paper, an area-efficient implementation of a fast converging square root algorithm is presented. The design of special arithmetic operations differs in many ways from the traditional tasks that digital designers are used to, and the role of parameterizibility and mapping of mathematical algorithms into digital hardware is discussed. Certain real-world applications requiring the use of the square root operator are presented, and it is argued, that implementing special arithmetic operations directly in hardware offers significant speed advantages over the conventional approach of implementing them in software. The mathematical algorithm of the square root operator is described, and its applicability to an implementation in digital logic is presented. It also is shown, that the the square root operator can be efficiently implemented without the need to resort to multiplications or divisions, which is advantageous in terms of both area and timing.

I. INTRODUCTION

Field programmable gate arrays (FPGAs) have been mostly used for Application Specific Integrated Circuit (ASIC) prototyping and implementing control logic. If the targeted application has been intensive in arithmetic operations —both in their number and variety— the designer has usually opted for either a microcontroller, a microprocessor or a signal processor. Choosing a processor-based solution to design problems has many obvious benefits, one of them being that the designer does not have to concern herself or himself with the actual implementation of the arithmetic operations. This decreases the crucial time-to-market and increases productivity, since there is no need to "reinvent the wheel".

However, there are cases when special attention needs to paid for the effective implementation of unusual arithmetic operations. A processor-based solution may be too slow and an ASIC-based solution may be too expensive. This presents additional problems for FPGA designers, who more often than not are not accustomed to implementing unsual arithmetic operations. Most FPGA synthesis tools provide efficient implementations for additions —which subsume subtractions and comparisons— that have been fine-tuned for the internal architecture and structure of the targeted FPGA devices, but when there is a need for special-purpose arithmetic operations, they must either be designed from scratch or acquired as an Intellectual Property (IP) block. The first option takes time and the second one costs money.

The mapping of mathematical algorithms into FPGA hardware is a work-intensive process, since the designer must be able to select the optimal algorithm satisfying multiple constraints. These include, but are not limited to, the convergence of the algorithm itself and the latency, throughput, required area and timing characteristics of its implementation. For example, if one algorithm has quadratic convergence but requires a lot of multiplications, a mathematically slower algorithm which can be implemented with shift operations instead of multiplications could be a better choice, especially if the area requirements are tight. Moreover, a good implementation of a mathematical algorithm should be parameterizable, which means that the algorithm should be coded in a hardware description language (HDL), of which VHDL and Verilog are the most obvious choices. All of the constraints mentioned above imply, that the designers of unusual mathematical operators must be well versed in both mathematics and the special features of the targeted hardware, which is a somewhat rare combination.

There are several FPGA features that can be taken advantage of when selecting and implementing mathematical algorithms for the computation of special arithmetic operations. Several FPGA device families have internal memory blocks, that can be effectively used to implement look-up tables. Another advantageous feature of programmable logic devices is that they are not inherently bound to any specified word-width. For example, if a particular application needs only 13 bits resolution, all arithmetic operations can be performed with exactly 13 bits resolution, which may imply substantial savings in the required area resources.

II. APPLICATIONS FOR THE SQUARE ROOT OPERATOR

Most of the research efforts into the hardware implementations of arithmetic functions have concentrated on the efficient implementation of adders and multipliers,

0-7803-5767-1/00/$10.00 ©2000 IEEE

which undoubtedly constitute a majority of the most frequently needed components of the arithmetic libraries. However, more specialized arithmetic functions are an indispensable part of many algorithms, and an efficient implementation of these arithmetic operations gives a big performance boost. In the following paragraphs, certain applications that may well benefit from an efficient hardware implementation of the square root function are briefly presented. It should be noted, that the potential uses of the square root function are by no means limited to the cases presented below.

In the case of genetic algorithms, non-linear functions are often needed in the evaluation of the fitness function. The hardware acceleration of genetic algorithms with reconfigurable logic has been able to speed up simulations by many orders of magnitude when compared to a software-based simulation [1]. An efficient implementation of the square root function widens the scope of available operation in fitness functions, and in that way makes a reconfigurable hardware-based solution for accelerating genetic algorithms more attractive.

In short, a simple genetic algorithm is based on the collective learning process within a population of indivuals, which is arbitrarily initialized. Each member of the population represents a point in the solution space of the problem, and by means of crossover, mutation and recombination new members are created. The fitness value of the new members is evaluated, and if deemed desirable, old members are replaced by more fit new members. A high-level block diagram of a simple genetic algorithm is presented in Figure 1.

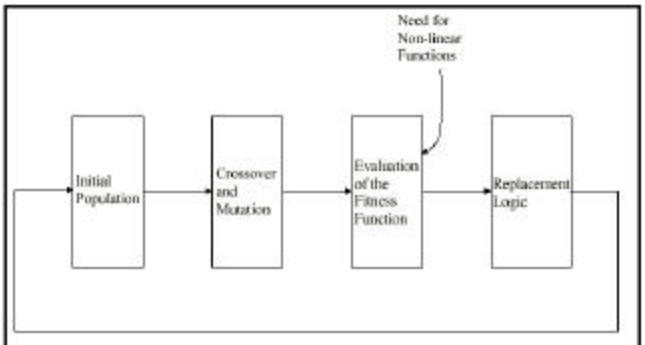

Figure 1 Block diagram of a simple genetic algorithm (GA)

The design of receiver blocks in communication systems benefits from an efficient implementation of the square root operator. As an example of a receiver block utilizing the square root operator, a block diagram of a noncoherent correlator, which is a part of an optimal noncoherent receiver in correlator form with equal symbol energies, includes a square root operator at its output [2] (See Figure. 2). It is obvious, that the more accurate and faster this operation is, the less bit errors result at the output of the receiver.

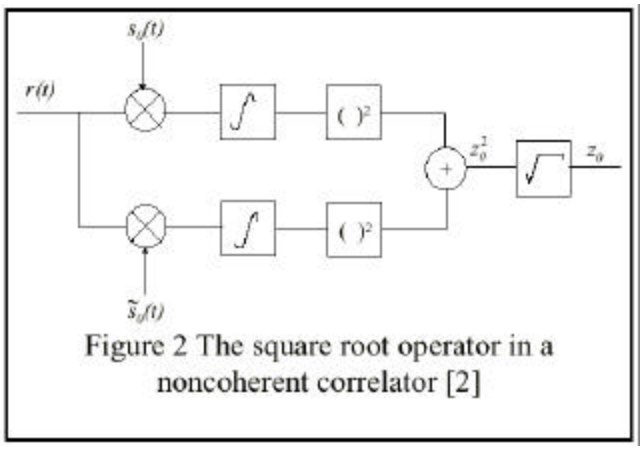

Figure 2 The square root operator in a noncoherent correlator [2]

III. THE SQUARE ROOT OPERATOR

The square root operation has been a somewhat neglected member in the arithmetic libraries of several microprocessors, and support for it has remained uneven. It has been noted, that microprocessor designers perceive square root —and division, to which it is closely related— as infrequent and low-priority operations, barely worth the trouble of implementing. A table describing the arithmetic performance of recent microprocessor floating point units (FPUs) in [3] presents illuminating results of this design philosophy. However, the efficient implementation of the square root operation is an integral part of many mathematical algorithms, and thus its relative rarity should not cover its obvious importance in many special cases.

A good overview of square root algorithms can be found in [4]. A description of the implementation of a non-restoring square root algorithm for single precision floating point numbers on FPGAs is presented in [5].

IV. ALGORITHM PRESENTATION

The implemented algorithm uses unsigned integers, which have several advantages over floating-point numbers in FPGA arithmetic. Operations on unsigned integers are often simpler to implement, and they require less chip area and resources. The square root operator assumes that its input argument has already been converted into an unsigned integer, which must be taken care of if an application uses signed integers.

The implemented algorithm is based on the bisection method presented by E. W. Dijkstra for approximating the square root of a given non-negative integer x [6]. More precisely, the algorithm finds a non-negative integer a satisfying the following constraints:

$$a^2 \leq x \text{ AND } (a+1)^2 > x \qquad (1)$$

The algorithm finds the square root of a *2n* bits wide number in *n* steps. The algorithm has been slightly modified from Dijkstra's original one, and therefore it will be called the modified Dijkstra's square root algorithm in this paper. The algorithm itself is represented in C language in Figure 3. The modified Dijkstra's algorithm does not need to perform multiplications, but instead two logical right shifts are used, which are not only very compact and convenient to implement in hardware, but also faster than multiplications by numbers which are not powers of two.

```
#define BITS 32

int square_root(int n)
{
  int mask = 1 << (BITS-2);
  int root = 0;
  int remainder;
  int i=1;

  remainder = n;

while (mask)
  {
    if ((root + mask) <=
remainder)
    {
      remainder -= (root+mask);
      root += (2*mask);
    }
  root >>= 1;
  mask >>= 2;
}

/* The following two lines of */
/* code should be left out if */
/* truncation is preferred to */
/* rounding.                  */

  if (remainder > root)
    root++;

  return root;
}
```

Figure 3 The modified Dijkstra's square root algorithm in C

The FPGA-tailored modified Dijkstra's square root algorithm starts by initializing three internal variables, *mask*, *root* and *remainder*, to MASK_INIT, 0 and *arg*, respectively. The individual bits in the constant MASK_INIT are set to zero except for the second most significant bit, which is set to one. For example, if the calculations are performed with 16 bits wide unsigned integers, the constant MASK_INIT equals 0100 0000 0000 0000. The internal variable *remainder* is set to the radicand, i.e. the number whose square root is to be calculated.

The modified Dijkstra's square root algorithm is executed for *n* steps when calculations are performed with *2n* bits wide numbers. Each step consists of a single comparison, where the sum of *root* and *mask* is compared to *remainder*. If *remainder* is larger than the sum of *root* and *mask*, the values of *remainder* and *root* are updated to *remainder-root-mask* and *root+2*mask*, respectively. During each step, *mask* is logically shifted right two bit positions and *root* is logically shifted right one bit position. The stopping condition is met, when *mask* equals zero. After the stopping condition is met, the following relation holds:

$$arg = root^2 + remainder \qquad (2)$$

If *remainder* is larger than *root* after the last step, *root* is incremented by one. This rounds up the result to the nearest integer. If truncation is preferred to rounding, this operation does not have to be performed.

V. FPGA IMPLEMENTATION

The modified Dijkstra's square root algorithm was coded in RTL VHDL and compiled and simulated with Altera's Max+Plus II version 8.2 design software. A separate VHDL package was written, where all parameters (for example, bit width and MASK_INIT) are defined. Because Altera's Max+Plus II v8.2 does not support VHDL shift operations, a logical right shift operation was also written and included in the VHDL package. A symbol of the top-level VHDL design entity of the modified Dijkstra's square root operator with parameterizable input argument width is presented in Figure 4.

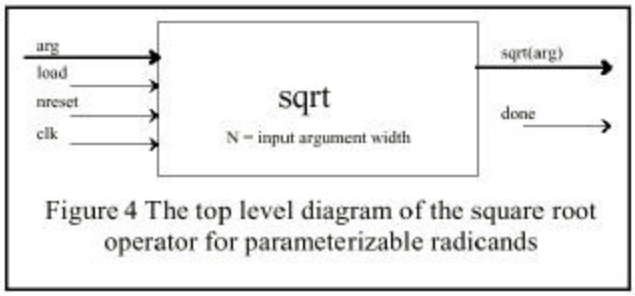

Figure 4 The top level diagram of the square root operator for parameterizable radicands

The modified Dijkstra's square root operator for 16- bit unsigned integers was compiled for Altera's 10K50 target device [7]. The device utilization and performance of the modified Dijkstra's square root operator is presented in

Table 1 .The latency of the modified Dijkstra's square root operator is only 8 clock cycles for 16-bit radicands.

Modified Dijkstra's square root operator	
Target Device	EPF10K50RC240-3
Device Utilization	
Logic Cells (LCs)	287
Embedded Array Blocks (EABs)	0
f_{MAX}	28.57 MHz
Latency (16-bit radicand)	8 clock cycles

Table 1. Performance of square root operator

VI. CONCLUSIONS AND FURTHER STUDY

An area-efficient iterative algorithm for the square root function was presented in this paper. The presented algorithm does not require multiplications, which is a considerable advantage when the algorithm was implemented on the programmable logic devices of Altera's FLEX10K family.

The importance of an efficient implementation of special arithmetic functions on FPGAs is often overlooked, since they are not frequently needed. However, certain applications that may directly benefit from special arithmetic operations were presented.

Further research goals include providing efficient floating-point support for the square root algorithms on FPGAs, finetuning the pipelining properties of the algorithm, and further study of additional special arithmetic functions and their implementation on FPGAs.

REFERENCES

[1] Matti Tommiska and Jarkko Vuori "Hardware Implementation of GA", *Proceeedings of the 2nd Nordic Workshop on Genetic Algorithm*s, Vaasa, Finland 19-23 August 1996, pp. 71–78.

[2] Stephen G. Wilson, *Digital Modulation and Codin*g, Prentice Hall 1996, p. 210.

[3] Peter Soderquist and Miriam Leeser "Division and Square Root: Choosing the Right Implementation", *IEEE Micr*o, Volume 17, Number 14, July/August 1997, pp. 56-66.

[4] Israel Koren, *Computer Arithmetic Algorithm*s, Prentice Hall, New Jersey, 1993, pp. 163–167.

[5] Yamin Li and Wanming Chu, "Implementation of Single Precision Floating Point Square Root on FPGAs", *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machine*s, April 16-18, 1997, Napa Valley, California, pp. 226232.

[6] Edsger W. Dijkstra, *A Discipline of Programmin*g, Prentice Hall 1976, pp. 61-65.

[7] *Altera Data Book 199*8, Altera Conrporation, 1998, pp. 29 - 30.