A Parallel/Vectorized Double-Precision Exponential Core to Accelerate Computational Science Applications *

Robin Pottathuparambil
Reconfigurable Computing Systems Lab
University of North Carolina at Charlotte
Charlotte, NC 28223–0001
rpottath@uncc.edu

Ron Sass
Reconfigurable Computing Systems Lab
University of North Carolina at Charlotte
Charlotte, NC 28223–0001
rsass@uncc.edu

ABSTRACT

Many natural processes exhibit exponential decay and, consequently, computational scientists make extensive use of e^{-x} in computer simulation experiments. While it is common to implement transcendental functions (sine, cosine, exponentiation, *etc.*) in hardware using the well-known CORDIC algorithm, many contemporary FPGA implementations either use fixed point or reduced precision floating-point operations (which suffers from a high average/mean error). Unfortunately, these solutions are unacceptable for many computational scientist who require the accuracy of double-precision values.

This paper presents a direct implementation of an IEEE 754 double-precision e^{-x} FPGA core to support computational science applications. The design is similar to CORDIC but has been modified to specifically support exponentiation; it is pipelined and parallel to efficiently handle large vectors of parameters. Compared to solutions described in the literature, it consumes lesser logical gates, enabling more e^{-x} cores per FPGA. The paper compares the implementation to the current prevailing approaches. Results shows that the implementation on the Virtex 4 XC4VFX60 FPGA achieves a correct precise double-precision e^{-x} values, with a high throughput.

Categories and Subject Descriptors

B.2.1 [**Hardware**]: Arithmetic and Logic Structures Design Styles [Parallel, Pipeline]; C.3.e [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems Reconfigurable hardware

Keywords

FPGA, CORDIC, Exponential core

*This project was supported in part by the National Science Foundation under NSF Grant CNS 06-52468 (CRI). The opinions expressed are those of the authors and not necessarily those of the Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA '09 Monterey, Californa USA
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

1. INTRODUCTION

From gravitational forces to gradients of protein concentration, many processes in nature exhibit an exponential decay property. Hence, computational scientists make extensive use of e^{-x} in computer simulations of physical phenomena. However, most microprocessors do not provide an exponentiation unit in hardware. Instead the operation is implemented in software via a combination of look-up tables, (floating-point) multiplies, and additions. These algorithms are very slow [4] and consume a large amount of energy when compared to hardware implementations. These drawbacks are magnified when the software implementation is used in computer simulation experiments on large, parallel computers.

Increasingly, Field-Programmable Gate Arrays (FPGAs) are being included in commercial high-performance computing systems as *compute accelerators* and a hardware implementation of e^{-x} would be an appropriate and advantageous use of these resources. Unfortunately, most hardware implementations of e^{-x} described in the literature are unsuitable for scientific applications. Most implementations either consume an unacceptable amount of resources (preventing multiple, parallel cores to be instantiated) or sacrifice double-precision IEEE 754 compatibility for speed (a trade-off that many computational scientists reject).

This paper describes an implementation and performance evaluation of a double-precision, IEEE 754-compatible parallel/vectorized exponentiation unit. At the heart of the design is deeply pipelined computation core similar to the well-known CORDIC algorithm. This core does not require large look-up tables and makes efficient use of the logic resources. Thus the high-level design can instantiate parallel cores resulting in an accurate, energy-efficient, high-throughput e^{-x} accelerator.

A twelve-way parallel implementation of the design has been synthesized for a Virtex 4 (XC4VFX60) device and tested on a Xilinx ML-410 developer board. Speed and power tests were conducted using a complete base system (DDR2 RAM, networking, and essential peripherals) running Linux 2.6. The test application runs as a user process and all measurements include the time to communicate parameters to and results from the hardware core.

Compared to other FPGA designs reported in the literature, the design described here is far more suitable for computational science applications. Although there are lower latency implementations in the literature, these designs compromise precision. The design described here uses eight guard bits resulting an average error of 0.047×2^{-53} which

is better than the results reported in the literature (and bit-for-bit the same as GNU C library's implementation). Moreover, this design has the highest throughput of the FPGA designs surveyed which is important for science applications where the e^{-x} computations are often independent. The design also outperforms current microprocessors. It is approximately 2× faster and requires a fraction of the power. The design can also generate e^{x}, without any major changes in the design there by reducing the cost of an additional floating-point divide.

The rest of this paper is organized as follows. The next section provides an overview of various techniques for computing e^{-x} and related FPGA work. The details of the proposed design are described in Section 3. In Section 4, the performance results are compared against competing solutions. The paper concludes with a brief summary and discussion of future directions.

2. BACKGROUND AND RELATED WORK

This section gives a brief idea of various implementation techniques that are often used to compute transcendental functions. Typical software methods include Taylor Series, Table interpolation method and dynamic table look-up method. Hardware methods for computing exponential functions include CORDIC based methods, table look-up methods and polynomial approximation. Also this section covers the existing implementations in the literature. Most of the literature use ulp (units in last place) as an error metric. In general, according to [9], a floating-point number that is represented as $\pm d_0.d_1d_2...d_{p-1} \times \beta^e$ (Where β is defined as the base, e is defined as exponent to the base and p is defined as the precision) If this floating point number is in error and is expressed as some z, then it is in error by $|d.dd...d - (z\beta^e)|\beta^{p-1}$ units in last place (ulp).

2.1 Taylor Series

A function represented as a sum of terms at a point is called as a Tylor series. These terms are calculated from the values of its derivatives. The Taylor series can be used to calculate exponential function e^x and trigonometric functions like sine and cosine. Some of the software techniques use Taylor series for the calculation of exponential function. Since Taylor series is a infinite sum of terms calculated from the values of its derivatives, the computational complexity of each term grows as the series grows. The implementation of such a algorithm becomes easy in software, however would take larger time to compute the values. This growing computational complexity of the series becomes a real bottleneck for implementation in hardware. Because of this, Taylor series is rarely used on its own. However partial sums of the series can be used as a good approximation for rounding off certain computed values. The partial sums so called as Taylor polynomials are combined with table method to get good rounded off values for the computation.

2.2 Look-Up Table Interpolation

Interpolation is a method of constructing new data points from a set of known data points. The known set of data points are obtained by sampling or from an experiment. These data points are then used to construct a function, which closely fits the data points. This process of constructing a function from a set of data points is called as curve fitting. Interpolation is a specific case of curve fitting where the function must exactly match the data points. A function which is too complex to be evaluated can be approximated by a simple function. This method is closely related to interpolation. The approximation is done by taking some data points from the complex function and creating a look-up table. The data from the look-up table is then interpolated to construct a simpler function. When a transcendental function is to be computed, the values for the function are stored in a look-up table and then are used for computing the values. Since the look-up table is a approximation of the function, the results are not accurate, but depending upon the accuracy needs, look-up table values can increased.

2.3 Exponential GNU library

The GNU C library (glibc) [15] provides a double-precision exponential function. This function is provided in the math library (libm.a) The library calculates $e^{(-x)}$ by applying commonly known mathematical identities such as,

$$e^x = 2^{x \cdot \log_2 e} = 2^{x_i} \times e^{x_f} \tag{1}$$

Where x_i is the integer part of $x \cdot \log_2 e$. The fractional part, $x_f = x - x_i \cdot \ln(2)$, is further processed to evaluate the mantissa. The fractional part is then again split into two or three independent sections and are evaluated separately by look-up table method. The size of the look-up table is 5.5625KB (uexp.tbl) with 712 double precision entries. A total of fourteen double precision multiplications (e_exp.c) are performed to generate the results.

2.4 Memoization (Dynamic Table Look-Up)

The term *memoization* comes from 1968 article in the journal *Nature* where a Memo function was described [16]. A memoized function is one which calculates results using a set of input parameters, and if the same function is called again with the same set of input parameters, the function just returns the stored values from the previous call rather than calculating it. The difference between the earlier described look-Up table and memoization is that look-Up tables are pre-computed; memoization populates the table as needed [18]. Memoization is a method by which the function's response time is reduced in exchange for the space. Memoization methods are used where one needs to trade off between space and time. When transcedal values are calculated for an application, the application may require the same values of the transcendental function again and again. The dynamic look-up table method or Memoization would basically generate a look-up table according to the calls made and when these calls are made again with same set of input parameters, the function just looks-up the table and returns the value. This would increase the space complexity of the application, but however if the application needs the same set of transcendental values then this method would be a good choice. This method is more dependent on the type of application.

2.5 CORDIC Implementations

COrdinate Rotation DIgital Computer (CORDIC) is a digit-by-digit method for computing transcendental functions It is also known as Volder's algorithm [22]. The CORDIC algorithm belongs to the family of "shift-and-add" algorithms. The CORDIC was developed in late 50's and was used where the hardware did not have floating-point multipliers. CORDIC implementation on such a hardware was

quite simple as the algorithm requires only simple operations like addition, subtraction, bitshift, and table look-up. The CORDIC also belongs to the set of iterative algorithms. The CORDIC has two modes of operation, the vectoring mode and the rotating mode. In the vectoring mode a given vector is rotated to the x-axis, while recording the angle required to make the vector rotation. On the other hand, in the rotating mode, the input vector is rotated to a pre-defined angle. The rotation is done in small steps or in each iteration. The direction of rotation in each iteration is decided such way that the given input vector slowly progresses towards the required angle. As soon the vector's angle is close enough to the required angle, the iteration is stopped. The accuracy of the results from the CORDIC depends on the number of iterations. If a double precision value is computed using the CORDIC, the number of iterations is at most 52 times. That is an n bit accuracy is achieved with at most n iterations. Each iteration involves operations like addition or subtraction, bitshift (divide by 2) and table look-up. Each iteration involves three equations, one for the angle, one for the x-axis and one for the y-axis. The angle iteration basically decides whether the vector has to be rotated clockwise or counterclockwise. Once the direction is decided then vector is rotated.

Since CORDIC uses just simple computations to perform rotation, these algorithms have been extensively used in many applications. An iterative based CORDIC processor is discussed by Andraka [2], where the architecture is obtained simply by implementing the three iterative equation in hardware. The hardware consists of registers, bit-parallel shifters. A more compact design using a bit serial arithmetic, which can work at a much higher clock rate than the equivalent bit parallel design is also implemented. The design needs to be clocked w times for each iteration, where w is the width of the data word. The latency of the hardware is so high that, if one needs a 64-bit data word CORDIC implementation, it would take thousands of clock cycles to generate the results and if a 64-bit IEEE 784 data word is used, then the above implementation becomes more complex. Boudabous [3] implemented a simple CORDIC algorithm on a Xilinx VirtexE FPGAs. However the design is only for a n-bit data and not for 64-bit IEEE 784 data word. Also the relative error is too high. An application that requires highly accurate values, relative error becomes a bottleneck. Valls [20], [21] used redundant arithmetic operators for the implementation of CORDIC algorithms. The implementation focuses on enhancing the performance of CORDIC using redundant arithmetic. However the paper concludes by saying that redundant arithmetic-based CORDIC methods are not suitable for implementation on FPGA.

A modified CORDIC algorithm is implemented by Kantabutra [14], where a low-precision arithmetic components to approximate high precision computations has been used. The approximation error is corrected very quickly and periodically so that high precision values are computed. However the implementation of the above modified technique on an FPGA becomes very tedious and consumes lot of digital logic. Chen [5] implemented a fast additive normalization method like the CORDIC method. The values of normalization factors can be directly calculated from the remaining terms without any computation. The convergence rate of the proposed method is exponential. The major hardware components in this implementation include a 83.84 Kbits ROM, five carry lookahead adders and four redundant multipliers. The design for a single precision exponential computation has a maximum error of 0.8735×2^{-24}. An improved version of Chen's [5] work was done by in [6], where the major analysis was done on a double precision exponential unit. The double precision exponential unit has a maximum error of 0.6602×2^{-54}. The unit is implemented using 24,302 gates with a effective delay of 40.11 ns.

2.6 Table Driven Implementations

The table-driven method or the look-up table method can also be used to calculate exponential values. As suggested by Tang [19], the implementation consists of three main parts. The input value is first reduced to a certain working range. A shifted exponential function is then estimated using known polynomial approximations. Finally, the exponential function of the original input is reconstructed using certain formula. The proposed error is accurate within 0.54 ulp as along as the final result does not underflow, in case gradual underflow, the error is till no worse that 0.77 ulp. The implementation of the above algorithm on to a platform FPGA becomes a challenging task.

A single precision exponential core was developed by Bui [4] that is in similar lines that of [19]. The algorithm was developed both in VHDL and Verilog. These designs are composed of numerous procedures that perform IEEE 754 operations. These operation include addition, multiplication and division by 32, rounding to nearest integers, modulo 32, comparison and powers of 2.The results show that outputs differ from the expected results by a small margin. However for application that require high margin of accuracy, the marginal error becomes a real problem. Doss [8] has also implemented exponential unit using Tang's [19] approach. However the implementation was a IEEE-754 single precision exponential unit. The implementation consumed about 30% of the available resources of Xilinx Virtex-II 4000 part for a fully pipelined version.

Vazquez [1] implemented a exponential function using floating point multiplier and additional hardware components. The computation is divided into three parts. The first part and third part is done using a table look-up and polynomial approximation. The second part is computed based on a transformation of slow radix-2 digit-recurrence algorithm into a fast computation by using a floating point multiplier and additional hardware. The use floating point multiplier would increase the complexity of design and would increase the number of gates. A fully-pipelined single precision exponential function consuming a small fraction of FPGA resources was implemented by Detrey [7], with a much lesser latency. The hardware implementation was targeted for a Virtex II XC2V1000-4 FPGA. The algorithm consists of shift operations, multipliers, range reduction operations and rounding operations. The algorithm is table-driven, fairly simple and straightforward, however designed for single-precision values.

2.7 Polynomial Approximation Methods

In the polynomial approximation method for the exponential calculation, a polynomial of a lesser degree is calculated that approximates $exp(x) - 1$. Harrison [10] implemented a exponential core in similar lines of polynomial approximation. The given value of x is reduced and a polynomial

is computed which approximates $exp(x) - 1$. The latency of the core is about 60 cycles with a maximum error of 0.51 ulp. Ernest [13] implemented a double-precision exponential function, which employs three independent look-up tables and short Taylor series approximation. The input argument after converting into a fixed point format is divided into a integer part, fractional part, fractional Taylor part. The integer is nothing but shifts and the fractional part is looked up in the table. The Taylor part is approximated to a polynomial of smaller degree. The proposed design consists of optimized multipliers, BRAMS, flip-flops and 4-input LUTs. The mean absolute error around 0.5 ulp with four guard bits.

3. DESIGN AND IMPLEMENTATION

3.1 Formulation

This section discusses the design of a CORDIC based hardware core that computes double precision e^{-x}. As defined by Hekstra [11], a floating-point CORDIC is one in which a vector is rotated over a angle, where all values are IEEE double precision floating point. The angle derived depends on whether the mode of operation is vectoring or rotation. The range of angle is confined to be between 0 and $\ln(e)$ i.e $x \in (0, \ln(e))$. The basic convergence range of the exponential function is between 0 and 1.1182 i.e $x \in (0, 1.1182)$ [12]. The convergence range is kept less than the maximum value so that precise results can be generated.

The paper concentrates on the rotation mode of CORDIC. The CORDIC method is governed, the angle z, x coordinate and the y coordinate equations. The x and y coordinate equations corresponds to cosh and sinh respectively. The difference of $\cosh(x)$ and $\sinh(x)$ for $x > 0$ is e^{-x}. The CORDIC equations for hyperbolic rotations are shown below.

$$x_{i+1} = x_i + y_i \cdot d_i \cdot 2^{-i} \quad (2)$$

$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i} \quad (3)$$

$$z_{i+1} = z_i - d_i \cdot \tanh^{-1}(2^{-i}) \quad (4)$$

where

$$d_i = \begin{cases} -1 & \text{if } z_i < 0 \\ +1 & \text{otherwise} \end{cases}$$

The value of d_i indicates the direction of rotation, x and y are the vector components. The CORDIC algorithm iterates to either zero y or zero z depending on the function required. The direction of vector rotation which is controlled by d, is used to achieve this. The above set of equation can be reduced to equation (5) as e^{-x} is the difference of $\cosh(x)$ and $\sinh(x)$ for $x > 0$. Also e^x is the sum of $\cosh(x)$ and $\sinh(x)$ for $x > 0$.

$$(x_{i+1} - y_{i+1}) = (x_i - y_i) - (d_i \cdot 2^{-i}) \cdot (x_i - y_i) \quad (5)$$

Let $(x - y)$ be w, then equation (5) can be re-written as

$$w_{i+1} = w_i - w_i \cdot d_i \cdot 2^{-i} \quad (6)$$

Hence there are only two equations (4) and (6) that needs to be iterated to calculate the value of e^{-x}. In the implementation, a signal is used to either compute e^{-x} or e^x. The bit signal overrides the minus sign in the equation (6) to a plus sign to compute e^x.

3.2 Top-Level Design

The overall top level system design is shown in Figure 1. Processor Local Bus (PLB), which is the main bus connects various IP cores like UARTLite, Ethernet, P/V (Parallel/Vectorized) Core, DDR2 (Double Data Rate RAM) and the PPC 405 core. The interrupt control block connects the interrupts from the various blocks (UARTLite, Ethernet, P/V Core) and generates a interrupt signal to the PPC405 core. The PPC Core then decodes to find the interrupting peripheral. The UARTLite block is used for Standard I/O (stdio). The Ethernet Block (LL Temac) is used for TCP/IP communication. The PPC 405 core runs at 300 MHz. A 2.6 version of Linux is configured for the above base-system and it runs out of DDR2. With such a configuration the base-system would be able to run MPI based applications. The P/V core block has multiple instantiation of the DEXP (Double Precision EXPonential) compute core, in which the $exp(x)$ computation is performed.

The P/V core has twelve parallel instantiation of the DEXP pipelined compute cores. These cores are instantiated whenever the input data is available. The P/V core has four major components, FIFO (First-in First Out), 1:8 De-Multiplexer, 8:1 Multiplexer and the DEXP Compute Core as shown in Figure 2(a). There are two FIFOs in the P/V core. The input FIFO and the output FIFO. The FIFOs are placed in the input and output to match the rates at which the data is coming in, the data being computed and the data being fed out. The FIFOs are 64 bit wide and 1024 elements deep, designed out of BRAMs. The demultiplexer handles the job of distributing the data from the input FIFO to each of the DEXP Compute Core (DCC). Each compute core can accept a burst of five elements. Once the computation is done the results are written to the output FIFO.

The input data for $exp(x)$ computation is computed in PPC 405 and is written into the DDR2 memory. A DMA transfer of data is done from the DDR2 to P/V Core. The first 64-bit (double-word) takes about 25 clock cycles and next word is available in the next clock cycle. The P/V core acts as a master core and bursts data from DDR2 to the input FIFO. A total of sixteen double-word is fetched from the DDR2 in one go. The data is then distributed among the DEXP Compute Core as shown in Figure 2(a). The fetched data is distributed in a pipelined fashion as shown in Figure 2(b) by the read process. The basic idea behind such a distribution, is to start the computation as soon as the data is ready. Once the computation is done the data is written to the output FIFO again in a pipelined fashion. This form of pipelined read and write to the FIFOs decreases the wait-for-data latency and improves the through-put of the entire design. Once the fetched data is distributed to all (twelve) of the parallel cores, the distribution process waits until any of the core is ready to do another set of computation. As soon as any one of the core has done the computation, the next set of input data (for exp computation) is fed into the core.

3.3 DEXP Compute Core

The DEXP(x) compute core (DCC) is a partially pipelined core. Since the core does a iterative-type computation, the pipeline needs to be flushed in every iteration. In any given iteration only n number of words can be computed, where n is the depth of the pipeline. In our case the depth is five and five elements (double-precision word) can be computed

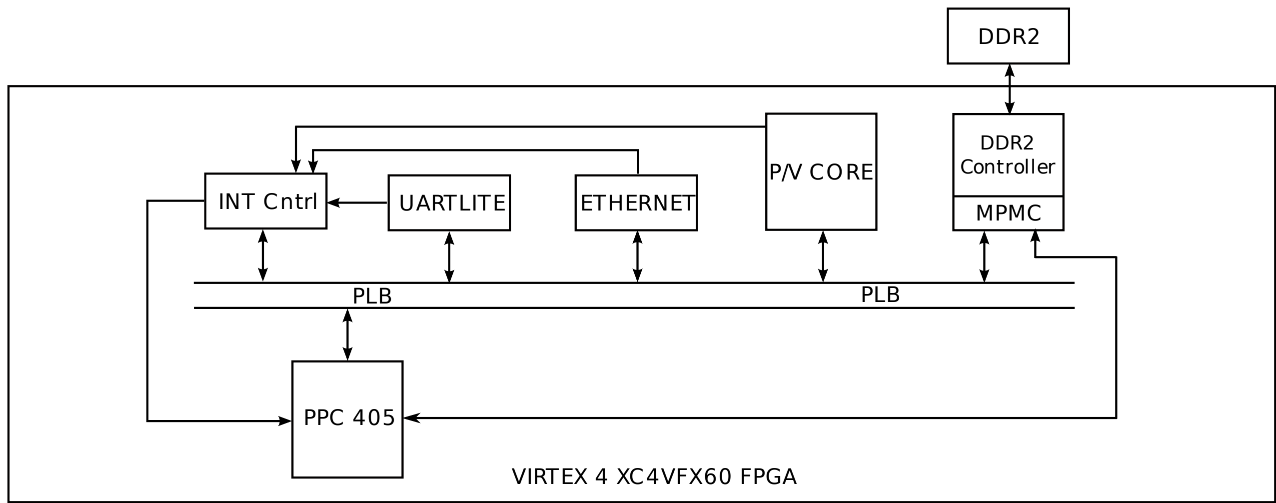
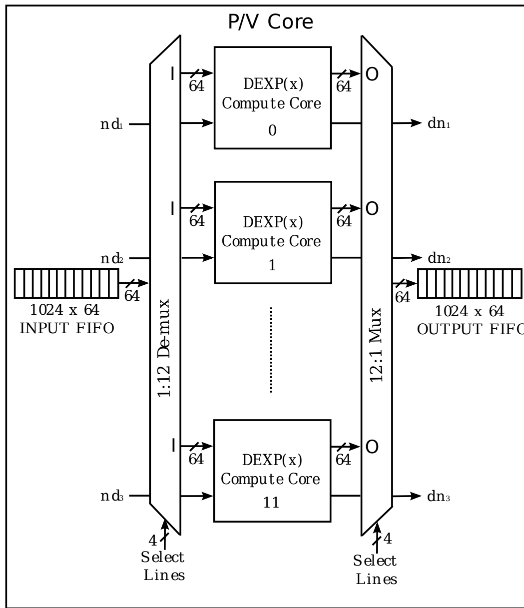
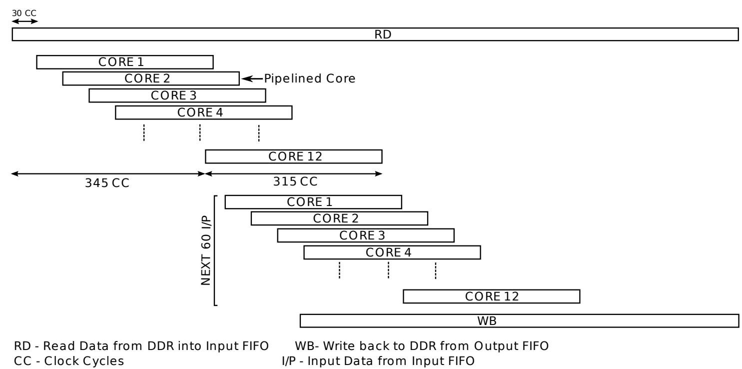


Figure 1: Design of Overall Base System





(a) (b)

Figure 2: (a) Overall Design of Pipelined-Vectorized System (b) Design of Overall Pipelined System

at any time. In the next iteration, the pipeline is flushed and the values of the previous iteration is fed into the pipeline.

The DEXP is computed using the equations (4) and (6) The two equations (4) and (6) has four major hardware components: Hyperbolic tangent look-up table, double precision adder/subtractor, 2:1 and 61:1 multiplexer and a bit shifter. The value of $\tanh^{-1}(2^{-i})$ is pre-computed for each i using a standalone 'C' code and these values are used to create a look-up table. These hyperbolic arctangent values are stored in the look-up table as IEEE 754 double-precision format with little modification. The most significant bits for the look-up table entries are the same and hence are stored as a 60-bit values instead of 64 bit. The look-up table entries depend on the number of iterations. The number of iterations is 61 in our design. These look-up table values are referenced through a 61:1 multiplexer. The look-up table requires about 456 bytes (60×61 bits). The four most significant bits are concatenated in the design logic, there by

reducing the size of the look-up table.

The double precision adder/subtractor is a four-stage pipelined core for addition or subtraction depending upon the operation flag. A 2:1 multiplexer is used to select the initial or the iterated value. A 61:1 multiplexer is used to select the hyperbolic tangent values from the look-up table. The values are selected using the iteration count. A bit shifter is used to replace the division by two operation. The implementation of left bit-shifter in IEEE 754 double precision format is done by subtracting the exponent by one and concatenating with the mantissa and the sign bit.

The core consists of two main components: angle computation hardware and the magnitude computation hardware as shown in Figure 3 all written in VHDL and described in detail below. The angle hardware module shown in Figure 4 is designed in accordance with the equation (4). For calculating a double precision e^{-x}, the iterations needs to done on a 64-bit wide data. The equation requires hyperbolic arctangent values for performing iterations. These values from the look-up table are then fed into the 61:1 multiplexer. The current iteration value is used to select the value from the look-up table, using the multiplexer. The multiplexer value is then added or subtracted from the previous result of the iteration. The result is then fed back for the next iteration. A 2:1 mux is used for choosing between the input angle value (x) and the previous iteration result. When its the first iteration, the input angle value is chosen. The result of each iteration step decides whether the next iteration is addition or subtraction operation. If a negative result is obtained then an addition is performed in the next iteration. The last bit of the result, which is the sign bit decides the next iteration addition/subtraction operation. This value is also used for the magnitude iteration hardware.

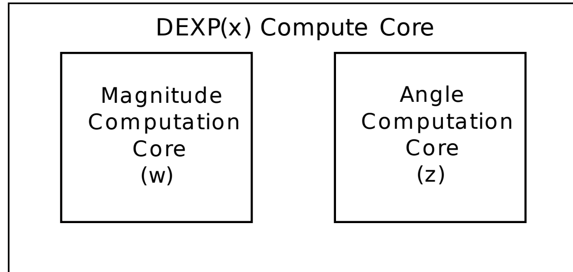


Figure 3: Design of Vectorized DEXP Compute Core

The magnitude iteration hardware module shown in Figure 5 is designed in accordance to the equation (6). works on an extended floating point format (72 bits). Eleven bit for exponent, sixty bits for mantissa and one sign bit. The initial value is fed into the subtractor or the 'Sub' block which does a subtraction on the exponent part of the double precision value. This corresponds to dividing a double precision value by 2^i. The mantissa and the sign bit is then concatenated in the 'Conc' block with result from the 'Sub' block. The value is then given into the floating point adder/subtractor. The addition/subtraction operation is dependent on the value of previous angle iteration. If the angle iteration gives a negative value in the previous iteration a

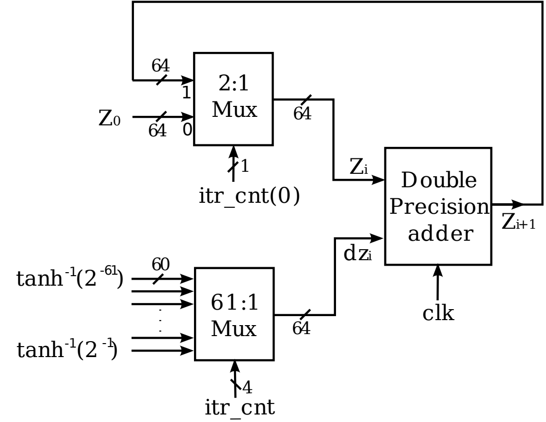


Figure 4: Design of Angle Computation Hardware

subtraction is performed or addition is performed. The result is then fed back for the next iteration. The initial value for the iteration is chosen by using a 2:1 mux. The elemental rotations in the hyperbolic coordinate system do not converge, However convergence can be achieved if iterations $4, 13, 40, 121.., k, 4k + 1$ are repeated [17]. Each hardware module is iterated n of times until a satisfiable accuracy is achieved. An $n + 1$ number of iteration produces a n bit of accuracy [20]. A total of 61 iterations are required for a precision of 52 bits with eight guard bits. In order to increase the accuracy of the results the iterations are performed on a 72-bit data. These additional 8-bits, called as the guard bits increases the accuracy of the results.

The process waits until a valid new data (x) is received (`nd = 1`) for computation of $exp(-x)$. Once a value for computation is received, the data is fed into the floating point adder/subtractor core untill the the floating point adder/subtractor core is full. In the fifth clock cycle, the results are generated by floating point adder/subtractor core, these results are used as inputs for the next iteration. In case of the angle iteration process the hyperbolic arctangent look-up table value is read and is kept ready and in case of magnitude iteration process the previous/initial value is divided by 2^i ('i' subtracted from the exponent) and is kept ready for the next state. Once the data is ready for the next iteration, the iteration count is checked and if the iteration count has not reached a pre-set value the data is fed back to the pipeline. If the iteration count has been reached the pre-set value, the process outputs the result and transitions into a idle state.

3.4 Hardware-Software Interface

The hardware-software interface is a very important interface to communicate data from the user program to the hardware. A Linux device driver is written to read/write data from the DDR2 memory and to access the P/V core's control and address registers. The control register says when

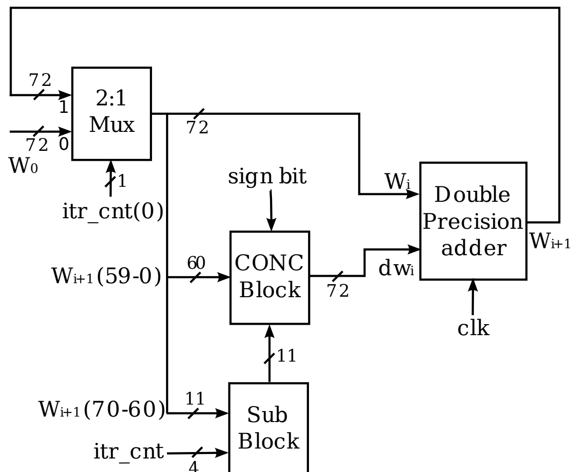


Figure 5: Design of Magnitude Computation Hardware

to start the computation and the amount of data to be computed. The 32-bit address register gives the starting location of the data. Once the user sets up the control register through the device driver and if P/V core is ready to compute $exp(x)$, it starts a DMA transfer from the DDR2 memory to the input FIFO. As soon as the Input FIFO has greater than five elements the P/V core distributes data to one of the free pipelined core and the computation is started. Once the output FIFO has sufficient data elements (sixteen data elements) a DMA transfer is again initiated to do a write back of the results to the DDR2 memory as shown in Figure 2(b). A minimum of sixteen data elements (results) are required to initiate a DMA transfer. As soon as the DMA transfer of all the computed data is done from the output FIFO an interrupt is generated from the P/V core. This generated interrupt, causes a PPC405 interrupt. This interrupt says the user that the computation is done.

4. RESULTS

The core described in the previous section has been implemented in VHDL. It was synthesized with Xilinx Design Suite 10.1 and Xilinx Embedded Development Kit (EDK). It was tested on an ML-410 board by comparing bit-for-bit with the GNU C Library output.

4.1 Design Analysis

This section deals with the various design parameters like slices, ROM table and the number of parallel instances. Before the design of the core a design analysis was conducted to find out what could be the slice count and there by the number of parallel instances. The design is a slight modification of the CORDIC algorithm. By carefully analyzing the equations (4) and (6) we came to a conclusion of having two double precision adder/subtractor core. These cores were generated from coregen. Coregen clearly gives the slice utilization with the maximum speed. Each of the double-precision adder takes about 600 slices with three DSP48 cores. However one of the double-precision adder/subtractor was a 72-bit wide (to accommodate 8 guard-bits) and hence did not use any of the DSP48 cores With a little of control logic and a LUT (look-up table is about 64 slices), each dexp compute core should take about 1500 slices and 3 DSP48s. The estimation was very close and the actual number of slices was 1620 slices with 3 DSP48s. The consumed slices are about 6% of the total slices. The design was intended to run MPI based applications. Hence a base-system with PLB, memory (DDR2), processor and UART was required so that Linux 2.6 can be ported on to the system. The base-system design took around 4000 slices (16% of the total slices). The rest of the slices were used to instantiate multiple dexp compute cores leaving 5% slices for future design changes. With this design strategy we were able to achieve twelve cores on a single die. The dexp compute core design is so modular that, it can be just changed to fit in any FPGA however coregen should be used to recreate the double-precision adder/subtractor core. The features of the dexp compute core are shown in Table 1. When the design was formulated, a highly precise dexp compute core was aimed. In order to achieve this, guard bits were employed. Each guard-bit addition reduced the probability of error (change in the last bit) by the order of 2^{-b}. Where b is the number of guard bits. We had chosen eight guard bits so that the probability of error (of the order of 2^{-53}) was 0.0039. A further increase in the number of guard bits would cause a small change in the probability of error, however consumes more slices. The table also compares the existing implementation. Implementation results conclude that the precision of the designed core is extremely high, and could be used for application which require high precision in calculations.

4.2 Performance Analysis

This section deals with the various performance parameters like maximum error, guard bits and latency for the core and compares these parameters with the existing designs in literature. The error analysis of the core was done in a range of angle between 0 and $\ln(e)$ i.e ($x \in (0, \ln(e))$. However this range can be extended by using simple mathematical identities. The core can take in a extending value which is basically a multiplication by 2^p which corresponds to p left shifts [3]. $\ln(2) \times 10^{11}$ test values were used to test the core. The test values were generated between the input range. The observed maximum error was 2^{-53}. However the average error was 0.047×2^{-53}, which is about 5% of the test cases. This shows that the results are very less prone to errors. The design uses eight guard bits to achieve such a high number of the precise results. The core basically iterates 63 number of times to get a accurate results. As discussed earlier, since the core requires a 52 bits of mantissa with 8 guard bits, the number of iterations are 60 and iterations 4, 13, 40 are repeated. Since 63 iterations are performed, the latency of the core is 63×5 clock cycles. Every iteration takes about five clock cycles. The total latency is about 315 clock cycles for five elements. There are twelve cores in the design and hence the throughput is approximately one DEXP computation in 5.25 clock cycles ($\frac{315}{5 \times 12}$). The performance results of the core is shown in table 2. Our design has a lower latency and a lower average error.

Table 1: Features of the Core (ND means no data available in reference)

Details	Our values	Chen [5]	Jamro [13]	Doss [8]	Detrey [7]
Style	CORDIC	CORDIC + TD	Table-Driven	Table-Driven	Table Driven
Precision	Double	Single, Double	Double	Single	Single
Slices	23455	ND	5000	5564	948
DSP48/MUL18	36	ND	0	ND	ND
ROM table size	3.57 Kbits	81.875 Kbits	108 Kbits	ND	ND
Error	2^{-53} (max)	0.8735×2^{-53} (max)	0.4708 (Mean)	ND	ND
Guard bits	8 bits	ND	4 bits	ND	5 bits
FPGA	Virtex 4 XC4VFX60	ND	Virtex 4 LX200	Virtex II 4000	Virtex II XC2V1000
Core frequency	100 MHz	ND	166 MHz	85 MHz	100 MHz

Table 2: Performance Results of the Core (ND means no data available in reference)

Details	Our values	Chen [5]	Jamro [13]	Doss [8]	Detrey [7]
Error	2^{-53} (max)	0.8735×2^{-53} (max)	0.4708 (Mean)	ND	ND
Guard bits	8 bits	ND	4 bits	ND	5 bits
Computation time	5.25 clk cyc (52.5 ns)	86 gate delays	27 clk cyc (448 ns)	ND	85 ns

A set of throughput measurements were conducted on various processors and was compared with our P/V core. A 'C' code was written to test with 512, 2048 and 4096 double-precision input words. Our base-system was ported with Linux 2.6. A device driver was written/loaded to communicate with the P/V hardware core. The same 'C' code was then used to test the P/V core. The results are shown in the figure 6. Where the x-axis is the number of DEXP operations and the y-axis is the actual throughput in DEXP operations/μsec. The throughput of the DEXP compute core outperforms all of the processor computation.

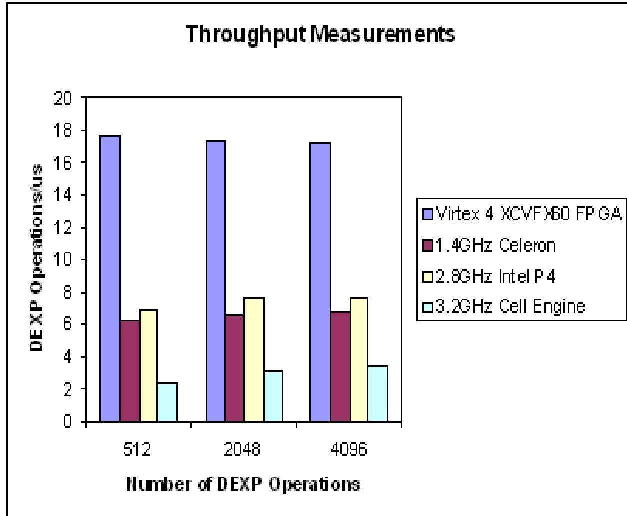


Figure 6: Throughput measurements of the Pipelined core

4.3 Power Analysis

A power analysis on the core was done for computing power starting from no DEXP compute core to twelve DEXP compute cores. A current probe was used to measure the current drawn by the lines from the SMPS (Switched Mode Power Supply (power supply in the cabinet)) to the FPGA. The power was calculated for each lines and summed up to find the total power used by the whole base-system and the ML410 board. A graph is plotted as shown in figure 7. There is a steady steep in the curve from no DEXP compute core to one compute core. This is because of the addition of the DMA logic and a single DEXP compute core. Also there is increase in power at two and four cores. This is because of change in the multiplexer and demultiplexer channel lines in the design logic. The power slope is steady from eight cores to twelve cores. The total power for the twelve way design is 19.85 W. The power consumption for no DEXP compute core, but with PLB bus, PPC, DDR2, BRAM, ML410 board is about 18.63 W. The average power per DEXP compute core is around 100mW.

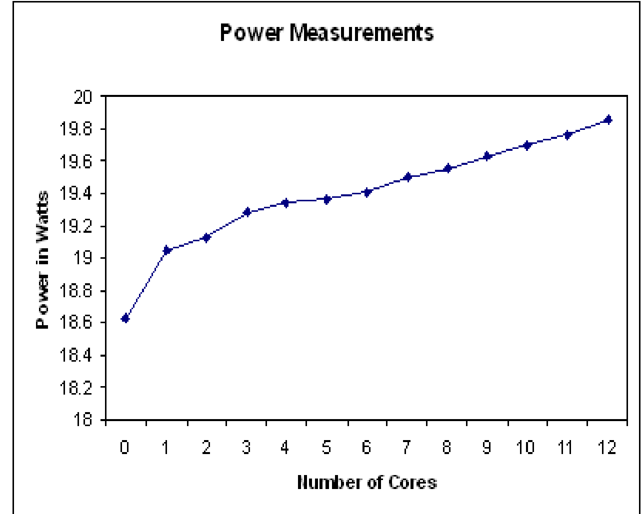


Figure 7: Power measurements of the Pipelined core

5. CONCLUSION

The paper describes a modified CORDIC core, for the implementation of double-precision exponential function on an platform FPGA. The core consumes only 93% of the total slices on Xilinx Virtex 4 XC4VFX60 device. The design described in the paper was able to instantiate a total of twelve cores with a complete base-system that runs Linux 2.6

out of the DDR2 memory. More parallel instantiation of the DEXP compute core was achieved by having a lesser logic cells for the design and a smaller look-up table. With such a design a high throughput of seventeen e^{-x} computations per μs (about 52.5 ns per e^x computation) was achieved. The core also generates e^x, without the cost of an additional floating-point divide. The maximum error of the designed core was 2^{-53}. The average error was 0.047×2^{-53} which constitutes only 5% of the test data. The higher throughput of the designed Parallel/Vectorized core brings in a lot of future work, in terms of accelerating large computational science applications.

6. REFERENCES

[1] Álvaro Vázquez and E. Antelo. Implementation of the exponential function in a floating-point unit. *J. VLSI Signal Process. Syst.*, 33(1):125–21, 2003.

[2] R. Andraka. A survey of cordic algorithms for fpga based computers. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 191–200, New York, NY, USA, 1998. ACM.

[3] A. Boudabous, F. Ghozzi, M. Kharrat, and N. Masmoudi. Implementation of hyperbolic functions using cordic algorithm. In *ICM 2004: Proceedings of the 16th International Conference on Microelectronics*, pages 738 – 741, 2004.

[4] H. T. Bui and S. Tahar. Design and synthesis of an IEEE-754 exponential function. In *1999 IEEE Canadian Conference on Electrical and Computer Engineering*, volume 1, pages 450–455, Edmonton, Alta., Canada, 1999.

[5] C. Chen, R.-L. Chen, and M.-H. Sheu. A fast additive normalization method for exponential computation. In *DSD '03: Proceedings of the Euromicro Symposium on Digital Systems Design*, page 286, Washington, DC, USA, 2003. IEEE Computer Society.

[6] C. Chen and K.-S. Cheng. An efficient exponential algorithm with exponential convergence rate. In *DSD '04: Proceedings of the Digital System Design, EUROMICRO Systems on (DSD'04)*, pages 548–555, Washington, DC, USA, 2004. IEEE Computer Society.

[7] J. Detrey and F. de Dinechin. Parameterized floating-point logarithm and exponential functions for fpgas. *Microprocess. Microsyst.*, 31(8):537–545, 2007.

[8] C. C. Doss and J. Robert L. Riley. Fpga-based implementation of a robust ieee-754 exponential unit. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 229–238, Washington, DC, USA, 2004. IEEE Computer Society.

[9] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[10] J. Harrison, T. Kubaska, S. Story, and P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, 1999 Q4, 1999. Available on the Web from `http://developer.intel.com/technology/itj/`.

[11] G. Hekstra and E. F. A. Deprettere. Floating-point CORDIC. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 130–137, Windsor, Canada, 1993. IEEE Computer Society Press, Los Alamitos, CA.

[12] X. Hu, R. G. Harber, and S. C. Bass. Expanding the range of convergence of the cordic algorithm. *IEEE Trans. Comput.*, 40(1):13–21, 1991.

[13] E. Jamro, K. Wiatr, and M. Wielgosz. Fpga implementation of 64-bit exponential function for hpc. In *FPL 2007: Proceedings of International Conference on Field Programmable Logic and Applications, 2007*, pages 718–721, 2007.

[14] V. Kantabutra. On hardware for computing exponential and trigonometric functions. *IEEE Trans. Comput.*, 45(3):328–339, 1996.

[15] R. McGrath. GNU C library, October 2007. Available on the Web from `http://www.gnu.org/software/libc/`.

[16] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[17] B. Parhami. *Computer arithmetic: algorithms and hardware designs*. Oxford University Press, Oxford, UK, 2000.

[18] S. Pope. Computationally efficient implementation of combustion chemistry using in situ adaptive tabulation. *Combust. Theory Modelling*, 1:41–63, 1997.

[19] P.-T. P. Tang. Table-driven implementation of the exponential function in ieee floating-point arithmetic. *ACM Trans. Math. Softw.*, 15(2):144–157, 1989.

[20] J. Valls, M. Kuhlmann, and K.K.Parhi. Efficient mapping of cordic algorithms on fpga. In *SiPS 2000: IEEE Workshop on Signal Processing Systems*, pages 336–345, 2000.

[21] J. Valls, M. Kuhlmann, and K. K. Parhi. Evaluation of cordic algorithms for fpga design. *J. VLSI Signal Process. Syst.*, 32(3):207–222, 2002.

[22] J. E. Volder. The cordic trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8, no. 3:330–334, 1959.