An Arithmetic Library and its Application to the N-body Problem

K.H. Tsoi, C.H. Ho, H.C. Yeung and P.H.W. Leong
{khtsoi,chho2,hcyeung,phwl}@cse.cuhk.edu.hk
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, NT Hong Kong

Abstract

Computer arithmetic is a specialist field of study, and it is very difficult for designers to choose the most efficient method for implementing a given algorithm due to the large number of design choices available. In this paper, an object oriented arithmetic library is presented which can be used to simulate and generate designs which use fixed, floating, logarithmic and hybrid number representations. The advantages of this approach are that a user can explore tradeoffs concerning precision, accuracy and speed from single high level description. Furthermore, users need not be intimately familiar with the implementation details of the underlying libraries, thus allowing users to develop systems employing advanced computer arithmetic without detailed knowledge of its implementation. The application of this library to a coprocessor which implements the force pipeline for an N-body solver is described.

1 Introduction

To date, field programmable gate arrays (FPGAs) have successfully been applied to speed up the computation of many fixed point problems in fields such as signal processing, pattern matching and cryptography. However, few scientific applications employing floating point or other number representations have been reported. We believe that this was due to limited FPGA resources restricting the amount of parallelism which can be obtained and the lack of high level computer aided design (CAD) tools and libraries to help in the design process.

Recent improvements in FPGA density have addressed the first issue to a certain extent. In order to address the second issue, a design methodology in which the arithmetic is generalized to be of arbitrary precision and representation is proposed. In this paper, a C++ based library, called the Computer Arithmetic Synthesis Tools (CAST), which supports arbitrary precision fixed, floating and logarithmic number systems is first described and then the application of this methodology to the N-body problem is given.

Although a wealth of knowledge about computer arithmetic exists, in practice the vast majority of custom computing machines (CCM) are made using the bit parallel two's complement representation. On current devices, it is possible to fit tens to hundreds of low precision floating point units on a single device and designs could possibly be made more efficient using floating point or other representations. Design productivity and the lack of standard numerical libraries to support different number representations appear to obstacles for the development of large scale scientific CCMs.

Module generators are able to generate customized designs from their input parameters, for example, the Xilinx LogiCore library for FPGAs [5] provides highly optimized libraries for the fixed point multiplication, multiply-accumulate, division and CORDIC operations and flexible floating point module generators have been developed [14]. In such libraries, there is usually little flexibility in the numerical representation which is usually fixed or floating point and there is no convenient way of using these blocks from a high level language apart from generating the component and then instantiating them as an element in the design.

PamDC [23] was one of the first module generators which allowed programmed generation of circuits from C++ and a recent extension, PAM-Blox II [18] has been reported. The underlying design of CAST was inspired by JHDL [11] in which circuits are treated as objects. Object oriented features of the C++ programming language are used to perform simulation and generation of synthesizable VHDL code by direct execution of the program. On top of this environment, a module library which provides a computer arithmetic scheme that is independent of numerical representation, number format and operators is available. The underlying circuit description is a structural one built up from primitive elements.

The N-body problem is computationally intensive and

involves a large number of arithmetic operations on numbers with large dynamic range. This together with the fact that relatively low precision is required makes it a good candidate for hardware acceleration. The most well known family of application specific integrate circuit (ASIC) based CCMs for the gravitational N-body problem is the GRAPE (GRAvitational PipE) computer [16]. GRAPE-4 was a winner of the IEEE Gordon Bell Prize in 1995 and 1996 and GRAPE-6 a winner in 1999, 2000 and 2001. Programmable FPGA based implementations of GRAPE, i.e. AHA Grape [13] and PROGRAPE-1 [9] have also been reported. Using the CAST tool, an FPGA based processor for the gravitational N-body problem similar to GRAPE, with the additional advantages of being flexible in the choice of arithmetic system and precision, was developed.

The contributions of this paper are as follows:

- The CAST tool takes a single structural description of the computation to be performed and can generate many different implementations with differing area, precision and performance requirements. This allows tradeoffs between the different designs to be quantified much more easily than with previous approaches.

- A module generator in which the arithmetic system is allowed to vary is presented. This has the advantages of facilitating better exploration of the design space, improving designer productivity through reuse and allowing for the encapsulation of the design details of the arithmetic system in libraries so that designers are able to apply them without detailed knowledge of the implementation.

- Although custom computing machines have been proposed for scientific applications, FPGA devices have only recently become large enough for practical scientific applications. The application of this methodology to the design of a force pipeline for the N-body problem demonstrates the flexibility of the approach to an important scientific problem and allows tradeoffs between area, precision and speed to be made in a quantitative manner.

The remainder of the paper is organized as follows. In Section 2, a description of the CAST system is described. In Section 3, the N-body problem is defined and the implementation of an FPGA based coprocessor for this problem is presented. Implementation results for the arithmetic library and N-body coprocessor are given in Section 4 and conclusions are drawn in Section 5.

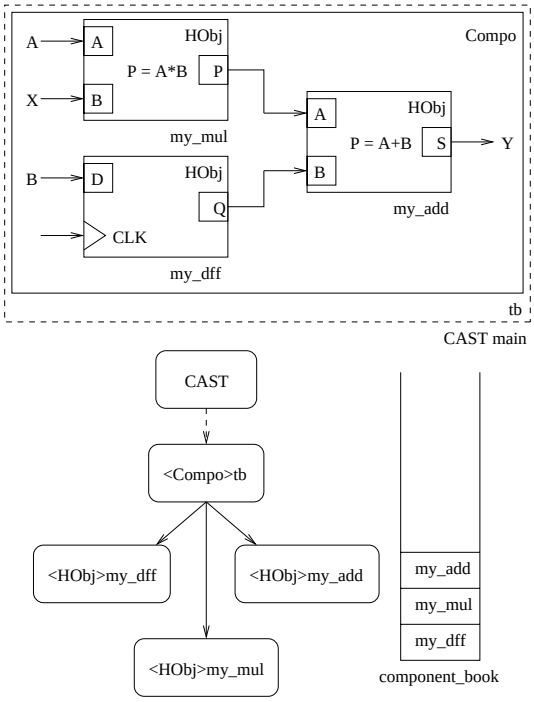


Figure 1. Example circuit and the object hierarchy.

2 Computer Arithmetic Synthesis Tool (CAST)

2.1 Framework

Two libraries are used in CAST. One is a utility library which is responsible for simulation and rendering of the circuits. The other is a primitive module library which consists of logic gates, adders, multiplexers, registers, etc. These can be connected together to form arbitrary designs and a circuit is modeled as a graph of interconnected objects. An example of a design to compute $y = ax+b$ is given in Figure 1. In this example, the testbench module *tb* includes three primitive modules: *my_mul*, *my_dff* and *my_add*. A component booker, also shown in the figure, is responsible for logging the creation of all primitives. In the object hierarchy, the composite module *tb* is called by the top level CAST system and is the parent of all three submodules. When the parent is to be simulated/rendered, all underlying children are simulated/rendered automatically.

2.2 Simulation and VHDL Generation

Two methods are used to simulate a circuit: `sim_clk()` for registering values at clock edge; and `sim_eva()` for

the combinational parts of the circuit. For primitive modules, the `sim_eva()` method is a set of expressions relating the outputs to the inputs. The `sim_eva()` method in a composite module calls `sim_eva()` of the submodules iteratively according to dependencies derived from the interconnection graph. When `sim_eva()` returns, the circuit is in a stable state and the value of any intermediate signal can be examined.

The simulation function of the CAST system helps designers to debug logic at a software level in the early stages of development. For primitive modules, it is the library designer's duty to ensure the simulation behaves the same as that of the generated VHDL circuit. CAST will ensure the consistency between the simulation and implementation for designs formed from an interconnection of primitives. Writing a testbench is also easier since the stimuli can be created using standard C++ functions.

The following example creates the adder object of Figure 1, perform a simulation and generate a VHDL description and testbench. "*my_add*" will be the instance name of the adder and the ports *A*, *B* and *S* will be generated automatically.

```
// create adder
my_add=new Add_n("my_add",2*n);
// connect I/O
connect(my_mul->P, my_add->A);
connect(my_dff->Q, my_add->B);
...
// simulate 1 clock cycle
tb.sim_clk();
// print out result
sim_result(add->S);
// generate VHDL (including testbench)
tb.gen();
```

When a module is created, the constructor first saves a local copy of the configuration, e.g. the adder width $2n$. Then the `circuit()` method is called to construct the circuit. Finally, the current object is registered to its parent.

To generate the VHDL code for a circuit, the `gen()` method is used. In this method, the I/O ports are first created, and then the components, their instances and interconnections are generated in a manner which avoids forward references.

2.3 Representation

In this subsection, a brief review of the fixed, floating and logarithmic number representations is presented. More detailed descriptions can be found in computer arithmetic textbooks such as Koren [12], Flynn [24, 8], Parhami [21] or Ercegovac [17].

Unsigned integers are used to represent the nonnegative integers. An N-bit unsigned integer has a range $[0, 2^N - 1]$ and can be described in binary form, with u_i being the i'th binary digit:

$$U = (u_{N-1}u_{N-2}\dots 0),\ u_i \in \{0, 1\}.$$

This represents the number

$$U = \sum_{i=0}^{N-1} u_i 2^i.$$

The two's complement representation is the most widely used scheme for integers. The representation is similar to the unsigned integers except that the most significant bit has a weighting of -2^{N-1}. A two's complement integer X of different N can be represented in binary form, with x_i the i'th binary digit as

$$X = (x_{N-1}x_{N-2}\dots 0),\ x_i \in \{0, 1\}.$$

X has a range of $[-2^{N-1}, 2^{N-1} - 1]$ and represents

$$X = -x_{N-1}2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i$$

The two's complement integer representation can be generalized to represent fractional numbers by scaling. A two's complement fraction is represented as a pair $(N, F)_{\mathcal{I}}$, where N is the wordlength, F is the fractional wordlength and the subscript $\mathcal{I}$ shows that it is an integer representation. The most significant $N - F$ bits of the number represent the integer part and the remaining F bits are the fractional part of the number

$$Y = (\overbrace{a_{N-1}\dots a_F}^{\text{integer}}\overbrace{a_{F-1}\dots a_0}^{\text{fraction}}).$$

This corresponds to a scaling of the two's complement integer representation by the factor $S = 2^{-F}$ and the two's complement fraction number Y represents

$$Y = 2^{-F} \times (-x_{N-1}2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i)$$

Note that the two's complement fraction $(N, 0)_{\mathcal{I}}$ corresponds to the two's complement integer case and $(N, N)_{\mathcal{I}}$ has a range of $[-1, 1)$.

Floating point numbers are an approximation to the real numbers and offer wider dynamic range than fixed point numbers, at the expense of reduced precision and larger implementation complexity and area. In the IEEE 754 standard [2] format, three fields are used to represent a floating point number and it can be represented as the pair $(N, F)_{\mathcal{F}}$

where N is the total wordlength, F is the wordlength of the significand (also known as the mantissa) and the subscript $\mathcal{F}$ shows that the pair represents a floating point number. The most significant bit is a sign bit A, the following $J(= N - F - 1)$ bits, b_i encode the exponent field B and the remaining F bits c_i encode the mantissa field C

$$Z = (\overbrace{a_0}^{\text{A}} \overbrace{b_{J-1} \ldots b_0}^{\text{B}} \overbrace{c_{F-1} \ldots b_0}^{\text{C}}).$$

A represents the sign S where

$$S = \begin{cases} +1 & \text{if } a_0 = 0 \\ -1 & \text{if } a_0 = 1 \end{cases}$$

The unsigned integers B and C are encoded representations of the exponent and mantissa respectively. The exponent E, is stored in a biased representation with $E = B - (2^{J-1} - 1)$. For normalized numbers, $B \neq 0$ and the significand is represented by $M = 1 + C \times 2^{-F}$. This is a two's complement fraction $(F + 1, F)_{\mathcal{I}}$ with the most significant bit being implicitly set to 1. If $B = 0$, it is called a denormalized number, and there is no implicit 1 in the $(F, F)_{\mathcal{I}}$ fraction.

The number represented is given by

$$Z = \begin{cases} S \times 2^E \times M & \text{if } (0 < B < 2^J - 1) \\ S \times 2^E \times (M - 1) & \text{if } (B = 0) \\ S \times \infty & \text{if } (B = 2^J - 1 \text{ and } C = 0) \\ NaN & \text{if } B = 2^J - 1 \text{ and } C \neq 0). \end{cases}$$

The logarithmic number system (LNS) is a special case of floating point in which the mantissa is always 1 (i.e. only the sign and exponent fields are used). It has the advantages of simplified implementation at the expense of reduced precision. For an N bit LNS number, $(N, F)_{\mathcal{L}}$, the most significant bit is a zero flag, Z. Z is zero if the number is zero (since there is no log of zero), otherwise set. The next most significant bit is used for a sign bit and the rest of the number is the base 2 logarithm of the magnitude of the number to be represented in $(N-2, F)_{\mathcal{I}}$ two's complement fraction format. If E is the value of this two's complement fraction and S is defined as for floating point, then

$$L = \begin{cases} 0 & \text{if } Z = 0 \\ L = S \times 2^E & \text{if } Z = 1 \end{cases}$$

2.4 Arithmetic Operator Library

CAST was designed to be extensible with a view that it can be used to support many different number systems, arithmetic operators and implementation schemes. In the current prototype, the fixed point, floating point and LNS number systems can be used and the operators supported are addition, subtraction, multiplication and $x^{-3/2}$, those being required for the N-body problem.

The implementation of the $+$, $-$ and $\times$ operators for the fixed point system follow the standard two's complement integer methods. A standard ripple carry adder/subtractor using the fast carry chain was used for addition. Different addition schemes such as carry select and carry lookahead for large wordlengths can be integrated into the CAST system by overriding the `gen()` function of this operator.

The input/output format and precision of the addition/subtraction fixed point operators are the same and no pre/post-processing is required. In the case of multiplication of two $(N, F)_{\mathcal{I}}$ two's complement fractions, an $(2N, 2F)_{\mathcal{I}}$ result is obtained. In CAST, the operators default to using the same format for inputs and outputs and so in order to convert the result back to $(N, F)_{\mathcal{I}}$ format, it must be scaled by 2^{-F} and the least significant N bits used.

The floating point operators are implemented in a manner similar to the IEEE 754 standard [2] except that not a number (NaN) and denormalized numbers are not implemented. The round-to-nearest mode is used for all operations and the size of exponent and fraction can be parameterized.

The floating point adder accepts two inputs $f1$ and $f2$ and returns the sum in the same format. The implementation is pipelined with a latency of 3 cycles. In the first cycle, $f1$ and $f2$ are swapped if the exponent of $f1$ is smaller than that of $f2$, and the difference between the exponents of $f1$ and $f2$ are calculated. In the second cycle, the significands are aligned. the intermediate sum is computed and the position of the leading one is determined using a priority encoder. In the final cycle, the result is normalized and rounded and the exponent corrected to produce the output.

The floating point multiplier accepts floating point operand $f1$ and $f2$ and return the product of operand in the same format as the inputs. In the first cycle, the sign bit is calculated and the intermediate exponent and product are also computed. In the second cycle, the intermediate result is normalized. In the third cycle, the result is rounded to produce the output.

The LNS implementation used in CAST is based on the open source code of the Aremaire project [7]. The LNS operations accept and produce numbers in the format described in Section 2.3. The multiplication in LNS is performed by summing the two exponents and setting the zero flag appropriately. The sign bit is computed as the XOR of the sign bits of the two inputs as in the floating point case. The LNS addition of $X = \log_2(x)$ and $Y = \log_2(y)$, `ADD_l`, is computed by making use of the following identity [12]:

$$\begin{aligned} Z &= \log_2(x \pm y) = \log_2\left(x(1 \pm y/x)\right) \\ &= \log_2(x) + \log_2(1 \pm 2^{\log_2(y/x)}) \end{aligned}$$

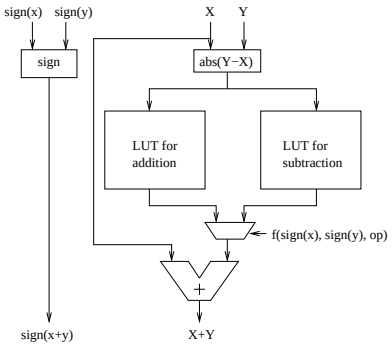


Figure 2. Simplified datapath of the LNS addition operation, `ADD_1`.

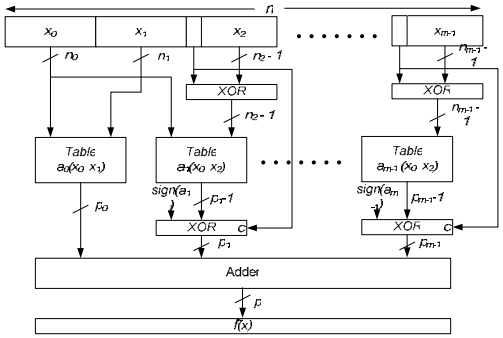


Figure 3. STAM datapath.

$$= \quad X + \log_2(1 \pm 2^{Y-X})$$

The implementation uses $Y - X$ to index a lookup table which generates $\log_2(1 \pm 2^{Y-X})$, and this table is constructed in Xilinx devices using distributed 16×1 LUT RAM rather than BlockRAM. When the y input is negative, a subtraction must be performed and thus the `ADD_1` module must include tables for both $1 + 2^{\log_2(Y-X)}$ and $1 - 2^{\log_2(Y-X)}$. Figure 2 shows a block diagram of the datapath for the LNS addition operation. In the actual implementation, extra swapping logic is included for the case that $Y - X$ is negative. To perform a subtraction, the sign bit of the second input is inverted prior to being passed to the addition module.

A class implementing the Symmetric Table Addition Method (STAM) [22], which can approximate any twice differentiable function is available to construct operators such as $x^{-3/2}$ [10]. STAM offers very good flexibility, however, the tables can become large if high accuracy is required. The datapath of the STAM implementation is shown in Figure 3. The input number is divided in several segments which are used as indices into the lookup tables. The outputs of these tables are summed and rounded to form the final result. The STAM algorithm requires large lookup tables if accurate function approximation is desired. These tables were implemented using the 18 Kbit BlockRAMs available in Xilinx Virtex-II FPGAs.

Computing the function $x^{-3/2}$ in LNS is done using shift and add operations to multiply the LNS number by -1.5. The fixed point implementation is computed directly using STAM. For floating point, STAM can only be directly applied to the significand part of the number. If the number is represented by $x = (1.f) \times 2^E$ where f is the fraction and E is the exponent, the floating point case can be handled using [10]:

$$f(x) = x^{-3/2} = ((1.f) \times 2^E)^{-3/2} = (1.f)^{-3/2} \times 2^{-3E/2}$$

A fixed point STAM module for $x^{-3/2}$ is used to calculate $(1.f)^{-3/2}$. If the exponent E is even, multiplication by $2^{-3E/2}$ can be achieved by simply multiplying the input's exponent by $-3E/2$. If E is odd, $x^{-3/2}$ can be rewritten as: $(1.f)^{-3/2} \times 2^{-(\lfloor -3E/2 \rfloor + 1)} \times 2^{-1/2}$. In [10], a floating point multiplication was used to handle the exponent of the odd exponent case. In the current design, a fixed point multiplier, as shown in Figure 4, was used to optionally multiply by $2^{-1/2}$ and the $2^{-(\lfloor -3E/2 \rfloor + E_0}$ term (where E_0 is the least significant bit of E) is added to the exponent. The new scheme results in a more compact circuit and eliminates the need for a normalization step before floating point multiplication. To improve throughput, pipeline registers were inserted and a 3 clock cycle latency introduced.

A set of modules for converting between number systems was also developed. When converting from floating to fixed point number systems, a shift amount is computed from the exponent. The fractional part (and the implicit '1' of the significand) will be shifted according to the shift amount. The final result should be two's complemented if the sign bit is set. When converting from fixed to floating point, the absolute value of the number is passed to a priority encoder to find the position of the most significant set bit. Then the number is shifted to form the significand and the exponent calculated. For conversion from LNS to the floating point system, the significand, $2^{frac(LNS)}$, where $frac(LNS)$ is the fractional part of the LNS number, is computed using a lookup table. The integer part of the LNS goes to the exponent after addition of the bias. In conversion from floating point to LNS, the integer part of the LNS is formed by subtracting the bias from the exponent. The fractional part of the LNS is computed by a lookup table of the $\log_2()$ function.

For all three number systems, operators may cause overflow/underflow. In the current hardware implementation, these special cases are not handled.

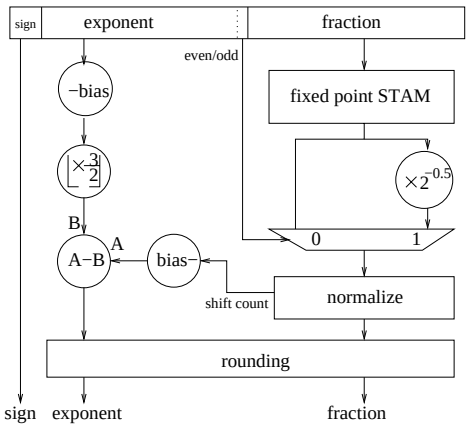


Figure 4. Floating point STAM datapath.

2.5 Unified Arithmetic Operator Class

A class of general arithmetic operators was developed. After the description of a circuit is constructed, the library provides an easy way to change the configuration of arithmetic operators in the circuit. Configuration of an operator includes the number system, the number format and latency allowed. This information is supplied as parameters when the object is created. For example, to use an 8-bit exponent and 23-bit fraction floating point adder with 3 clock cycle latency, the module is created as:

```
ADD_f("my_add", this, 8, 23, 3);
```

The operator interface for different number systems is unified in a single class: `CAST_ADD`, `CAST_MUL`, etc. The class includes operators from the parameterized fixed point, floating point and LNS libraries. As an example, the following code segment creates an LNS adder:

```
CAST_ADD("my_add", this, 8, 23, LNS);
```

Table 2.5 is a summary of the available arithmetic operators for the different number systems as well as the attributes bounded to these operators.

A latency parameter may be used to select different implementations. User can query the latency of any object using the `delay()` method. When different operators for different number systems and/or precision are used, their latency may change e.g. fixed and floating multipliers may have different latencies. When assembling a datapath, the user is responsible for matching the latency of the operators by inserting delay elements.

The unified operator class thus provides a consistent interface to the arithmetic library and encapsulates the internal details of their semantics and implementation in a manner that one can use the library with minimal knowledge about its implementation.

In the future, we would like to increase the usefulness of the arithmetic library in CAST by adding more number representations (e.g. redundant and residue number systems), arithmetic schemes (e.g. online arithmetic, division, square root etc), and incorporate existing libraries (e.g. the Xilinx LogiCore library, the UCLA Astra library for online arithmetic [6] and the floating point module generator in [14]) into its framework.

3 The N-body Problem

3.1 Description

A wide range of physical systems can be studied by modeling them as an N-Body problem. The N-Body problem is extensively used in various fields of science such as astrophysics [16] and molecular biology [19] In the N-body problem, particles are modeled as points in space and the evolution of the system of N particles is computed by solving a differential equation of the form:

$$\frac{d^2\mathbf{x}}{dt^2} = \sum_{j=1}^{N} \mathbf{F}(\mathbf{x}_i, \mathbf{x}_j) \quad (1)$$

where $\mathbf{F}(\mathbf{x}_i, \mathbf{x}_j)$ represents the force between particles i and j and is application dependent.

N-Body problems are solved using numerical integration in which the majority of the computation time is spent calculating $\mathbf{F}(\mathbf{x}_i, \mathbf{x}_j)$. Since the force calculation part is expensive and at the same time has a rather simple algorithm, the problem can benefit from hardware acceleration.

In the future, we intend to integrate the FPGA-based coprocessor with the NEMO stellar dynamic toolkit [1] which contains an implementation of Aarseth's nbody0 code. Aarseth's algorithm [3] uses the Adams-Bashforth-Moulton (ABM) predictor-corrector scheme to solve the gravitational N-body problem with time complexity $O(N^2)$. We also intend to use the hierarchical timestep algorithm [15] which reduces the amount of data transferred between the host and FPGA. In the algorithm, the positions of the particles are advanced in discrete timesteps. The length of timesteps are different for each particle and are dynamically adjusted for each particle according to how fast the force acting on a particle changes. A small timestep is needed to resolve a rapid change. In each timestep, the predicted positions of all the particles are computed. The forces acting each of the particles that are to be updated in the current timestep are then evaluated. Using the computed forces, the corrector equation is applied to those particles to update their positions. The process is then repeated to move the simulation forward in time.

Table 1. Summary of arithmetic operators available in the current CAST system.

	ADD	SUB	MULT	$x^{-3/2}$
Fixed Pt.	ADD_n() width	SUB_n() width	MUL_n() width	POWM15_n() width, segments, guard bits
Float Pt.	ADD_f() exp, frac	SUB_f() exp, frac	MUL_f() exp, frac	POWM15_f() exp, frac, segments, guard bits
LNS	ADD_l() int, frac	SUB_l() int, frac	MUL_l() int, frac	POWM15_l() int, frac
Unified	CAST_ADD_n() a, b, ns	CAST_SUB_n() a, b, ns	CAST_MULT_n() a, b, ns	CAST_POWM15_n() a, b, ns

Key - segments, guard bits: from configuration file according to width. **exp, frac**: width of exponent and fraction. **int, frac**: width of integer part and fractional part of exponent. **ns**: number system selection. **a**: width of fixed point, exp of floating point, int of LNS. **b**: frac of floating point, frac of LNS.

3.2 Coprocessor Implementation

An FPGA based coprocessor handling the force calculation part of the algorithm was built. The arithmetic core of the processor was generated from a C++ description using the CAST system. Since the accuracy requirement for different simulation runs can differ greatly and depends on the source data and the nature of problem being solved, being able to experiment with different wordlength and arithmetic systems facilitates better exploration of the design space.

The processor was design to work together with a host computer, which runs the NEMO N-body simulation code mentioned in the previous subsection. The host computer handles all computation except the force calculation. Particle positions are sent to the coprocessor board from a host processor through the board's interface. The coprocessor computes the force acting on a particle, i using Equation 2 where $\mathbf{x_i}$ and $\mathbf{x_j}$ are the position vectors of particles i and j respectively, $\mathbf{r}_{ij} = |\mathbf{x}_i - \mathbf{x}_j|$ and ϵ is the softening constant.

$$\mathbf{F}(\mathbf{x_i}, \mathbf{x_j}) = \sum_{j=1}^{N} \frac{\mathbf{x}_i - \mathbf{x}_j}{(\mathbf{r}_{ij}^2 + \epsilon^2)^{\frac{3}{2}}} \qquad (2)$$

The architecture of the implementation is shown in Figure 5. The main components are the control, particle memory and the force pipeline. The particle memory stores the predicted position of all particles while the force pipeline calculates the force acting one particle. In each timestep, the predicted particle positions are written to the particle memory by the host. For each particle i that is to be advanced in that timestep, the corresponding index is sent to the coprocessor. The corresponding particle position is then read from the particle memory and stored in a register. The force pipeline then begins the calculation as the positions of all j particles are retrieved and fed to the pipeline. The host polls the coprocessor to check if the calculation has completed and then reads the result from the coprocessor.

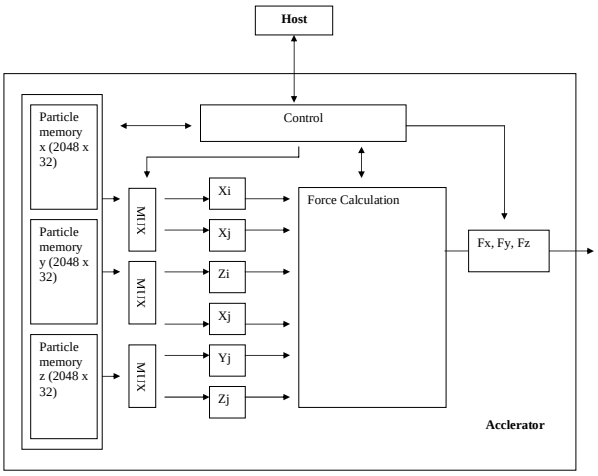


Figure 5. Top level block diagram showing the architecture of the coprocessor.

The force pipeline is the most critical part of the design. The speed of the pipeline directly affects the performance of the system. Figure 6 shows the datapath of the force pipeline. It is a fairly straightforward implementation of Equation 2 and is generated by the CAST system.

Although our implementation is similar in architecture to that of GRAPE-3 [16, 4], three features were not implemented in our design. Firstly, all the particles in GRAPE can be of different mass whereas our implementation assumes they are of the same mass. Secondly, GRAPE-3 calculates the gravitational potential as well as the gravitational force. In our integration algorithm, gravitational potential was not used and hence not implemented. Finally, GRAPE-3 has a neighbor function flag which is raised when two particles are closer than a certain amount.

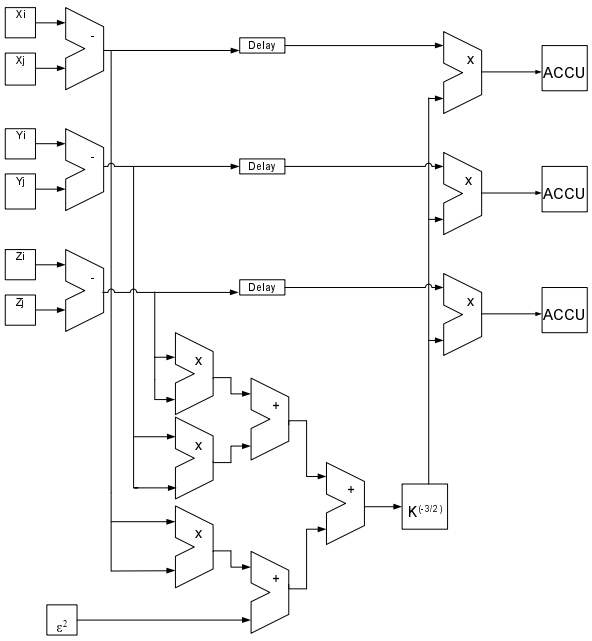


Figure 6. Architecture of the force pipeline.

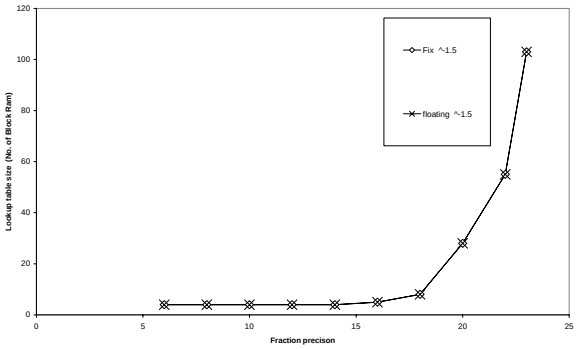


Figure 7. Memory usage of `ADD`, `MUL` and $x^{-3/2}$ (number of Virtex-II 18-Kbit BlockRAMs).

4 Results

In this section, results showing the resource utilization and performance of the individual operators in the CAST library, along with the precision and performance of the N-body coprocessor are presented. All of the results were simulated using both the CAST system in C++ and Synopsys VSS for verifying the generated VHDL. The target device was a Xilinx Virtex-II XC2V1000FG456-5 for all cases except those which required more than the 40 block RAMs available on that device. For those cases, namely the fixed and floating point implementations with a fraction size greater than or equal to 22, results for an XC2V4000-FF1152-5 are reported. Performance measurements are based on the reports from the Xilinx ISE 5.2i development tools.

4.1 Arithmetic Library

Three quantities were used to evaluate the performance of the operators: the maximum frequency as reported by the Xilinx tools, the logic resource utilization and the Block-RAM memory utilization.

The exponent wordlength of the floating point implementation and the integer part of the LNS system were fixed to be 8 bits in width. This configuration is similar to the IEEE 754 single precision standard and can operate without overflow in our simulations. For all 3 number systems, the `SUB` operations has similar performance to the `ADD` operation so they are not shown in the figure.

The number of BlockRAM memory resources required for the $x^{-3/2}$ operator are plotted in Figure 7. This is determined by the memory requirements of the STAM tables for both fixed and floating point implementations. As can be seen in the figure, since the floating point implementation uses the fixed point STAM for its significand, the memory requirements are identical. For the LNS implementations, $x^{-3/2}$ can be computed by multiplying by -1.5 and no memory resources were used.

The operating frequency and logic utilization are plotted against the number of fractional bits for different operators and number systems in Figures 8 and 9 respectively. These tables can be used to compare different implementations, precisions and numbering systems in the CAST arithmetic library, allowing a quantitative assessment of which approach is most suitable for a given application. Note that the LNS library [7] has a maximum LNS fractional wordlength of 13-bits and this limitation is carried over to CAST.

4.2 N-body Coprocessor

Using the CAST system, implementations of the N-body coprocessor with different fractional wordlengths using the fixed point, floating point and LNS number systems were made. The exponent wordlength of the floating point implementation and the integer part of the LNS system were fixed to be 8 bits in width.

In order to show the ability of CAST to deal with several number systems, an implementation, similar to GRAPE-3 [20] was built. In this hybrid format, a similar configuration as GRAPE-3 was used and thus a $(20, 10)_{\mathcal{I}}$ fixed point format was used to represent the position vectors of the particles and for calculating the difference between the

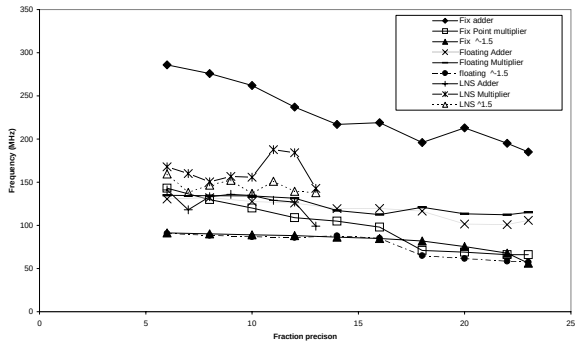


Figure 8. Frequency comparison of the `ADD`, `MUL` and $x^{-3/2}$ operators.

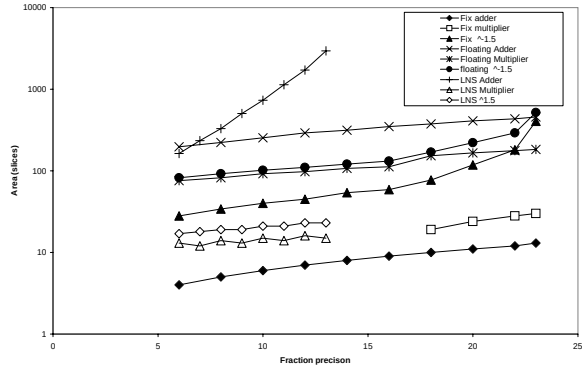


Figure 9. Area utilization of the `ADD`, `MUL` and $x^{-3/2}$ operators.

position vectors. The difference was then converted to a $(15, 6)_{\mathcal{L}}$ bit LNS format, which was used for all subsequent operations in calculating the partial force. The partial force was converted to a $(28, 28)_{\mathcal{I}}$ fixed point format which was accumulated to obtain the sum in Equation 2.

The implementations were simulated using the CAST system to evaluate their accuracy. To compare the precision of various implementations, the relative mean squared error $S_r(s)$, introduced in [4], was used. The relative mean square error measures the error in force calculation between a pair of particles and is defined as:

$$S_r(\mathbf{f}) = \frac{|\hat{\mathbf{f}} - \mathbf{f}|^2}{\mathbf{f}^2} \quad (3)$$

where $\mathbf{f}$ is the force computed by the hardware coprocessor

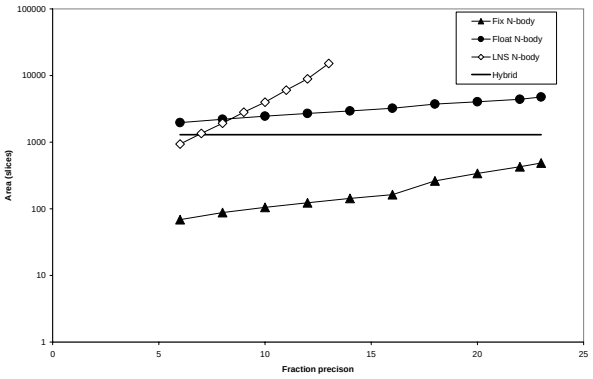


Figure 10. Area comparison of N-body implementations.

and $\hat{\mathbf{f}}$ is the reference value computed using IEEE double precision floating-point arithmetic. Since the relative mean square error depends on the distance between the 2 particles, pairs of particles with varying distance r was generated ($\epsilon = 0$) and the error computed. The resulting error function $\sqrt{S_r(s)}$ was plotted against r to obtain the error curves of Figure 12. The average error curve for GRAPE-3 [16, 4] is also shown for comparison. The fixed point implementation suffered from overflow for small r and underflow for large r due to insufficient dynamic range for this problem leading to large errors. Thus we do not consider fixed point to be a good representation for this problem.

A comparison of the area utilization for different numerical representations and fractional wordlengths is given in Figure 10. As expected, fixed point has the smallest area requirements. The LNS system has smaller area than floating point up to 8 bits, after which floating point is smaller, the main area overhead for the LNS lying in the addition operation which requires a large number of slices for large fraction sizes. The hybrid implementation has area between fixed and floating.

The reported maximum clock frequency for the different schemes is given in Figure 11. The fixed point implementation has the highest operating frequency and the floating point implementation is the slowest. The LNS and hybrid implementations achieve operating frequencies between the two.

If comparable accuracy to GRAPE-3 for the entire input range is desired, as mentioned earlier, the fixed point implementation is not suitable. This leaves the floating point $(21, 12)_{\mathcal{F}}$, LNS $(21, 11)_{\mathcal{L}}$ and hybrid implementations as candidates. By comparing their area and frequency requirements in Figures 10 and 11, it can be seen that the hybrid implementation offers a smaller area and higher frequency than the other two candidates. Thus, for the N-body example presented, based on considerations of precision, and

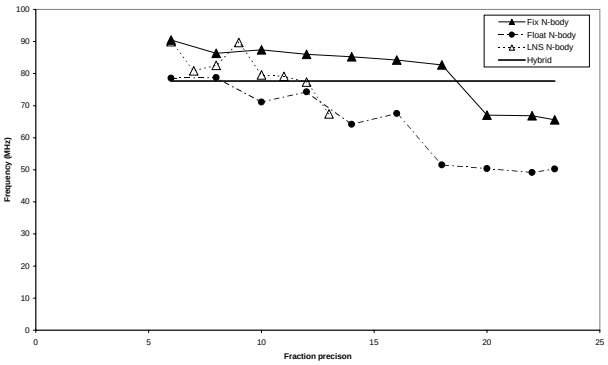


Figure 11. Performance comparison of N-body implementations.

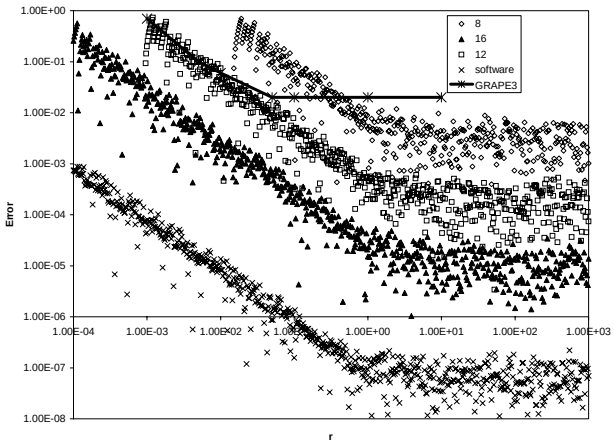
circuit area, the hybrid implementation appears to be the most suitable implementation scheme for the Xilinx Virtex-II XC2V1000FG456-5 device chosen. If, for example, a different device such as a Virtex device which does not have dedicated multipliers, were to be used, the tradeoffs may be different, and the same methodology could be used to aid in making the best choice.

5 Conclusion

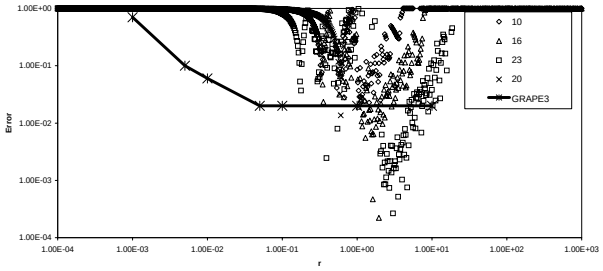
A design methodology in which the arithmetic is generalized to be of arbitrary precision and representation was proposed and a C++ based tool called CAST to aid in its implementation was described. CAST allows details concerning the simulation and implementation of number systems and operators to be captured within its framework. Users are then able to produce implementations and explore tradeoffs in area, speed and precision without requiring a detailed understanding of the internal implementations. CAST enables more efficient exploration of the design space to be conducted, reduces design time and reduces the amount of computer arithmetic expertise required to develop a system using a supported number representation.

The CAST system was applied to the problem of designing a coprocessor to compute the solution of the N-body problem. From a structural description of the computation to be performed, a large number of different designs were simulated in C++ and the corresponding VHDL code rendered, each implementation having different tradeoffs in precision, area and speed. By constraining the design to be of a certain precision, it was possible to determine the smallest fractional wordlength which could meet the accuracy criteria for the fixed point, floating point, LNS and hybrid implementations. A comparison of area and frequency suggested that the hybrid implementation was the best solution. Different constraints on precision, area and speed may

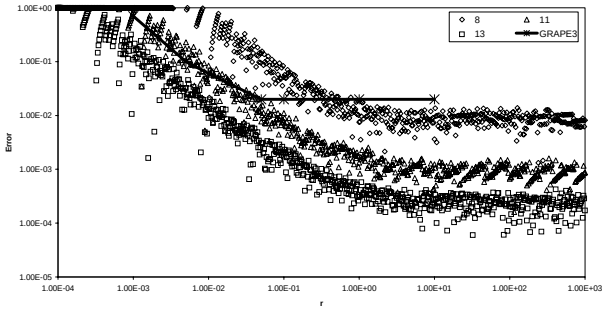
Figure 12. Quantization error for force calculation in the N-body problem.



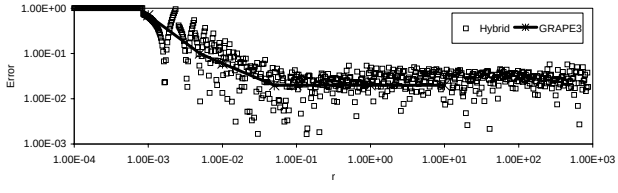
(a) Floating point quantization error.



(b) Fixed point quantization error.



(c) LNS quantization error.



(d) Hybrid quantization error.

lead to different choices, easily identified from the graphs obtained.

Acknowledgement

The authors gratefully acknowledge support from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4333/02E).

References

[1] NEMO - A Stellar Dynamics Toolbox. In *http://bima.astro.umd.edu/nemo/*.

[2] *IEEE standard for binary floating-point arithmetic: ANSI/IEEE std 754-1985*. 1985.

[3] S. J. Aarseth. Direct methods for n-body simulations. In *Multiple Time Scales*. Academic Press, 2001.

[4] E. Athanassoula, A. Bosma, J.-C. Lambert, and J. Makino. Performance and accuracy of a GRAPE-3 system for collisionless n-body simulations. In *Monthly Notices of the Royal Astronomical Society*, pages 369–380, Feb 1998.

[5] X. Corp. *Xilinx Core Generator*. http://www.xilinx.com/ipcenter/, 2002.

[6] M. Ercegovac, J. Pipan, and R. Mcllhenny. Astra: Arithmetic scripting tool for reconfigurable architectures. 2002. http://unagi.cs.ucla.edu/Astra.

[7] F. Dinechin. FPLIB. 2003. http://perso.ens-lyon.fr/jeremie.detrey/FPLibrary/.

[8] M. J. Flynn. *Advanced computer arithmetic design*. Wiley, 2001.

[9] T. Hamada, T. Fukushige, A. Kawai, and J. Makino. Prograpе-1: A programmable multi-purpose computer for many-body simulations. *Publ. of the Astronomical Society of Japan*, 52:943–954, 2000.

[10] C. Ho, K. Tsoi, H. Yeung, Y. Lam, K. Lee, P. Leong, R. Ludewig, P. Zipf, A. Ortiz, and M. Glesner. Arbitrary function approximation in HDLs with application to the n-body problem. In *2003 IEEE International Conference on Field-Programmable Technology (FPT)*, pages 84–91, Dec 2003.

[11] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting. A CAD suite for high-performance FPGA design. In *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–24, April 1999.

[12] I. Koren. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.

[13] T. Kuberka, A. Kugel, R. Manner, H. Singpiel, R. Spurzem, and R. Klessen. Aga-grape: Adaptive hydrodynamic architecture - GRAvity PipE. In *Field Programmable Logic and Applications*, volume LNCS 1673, pages 417–424, 1999.

[14] J. Liang, R. Tessier, and O. Mencer. Floating Point Unit Generation and Evaluation for FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 185–194, 2003.

[15] J. Makino. A modified Aarseth code for GRAPE and vector processors. In *Publ. of the Astronomical Society of Japan*, pages 859–867, Dec 1991.

[16] J. Makino and M. Taiji. Scientific simulation with special-purpose computers - the GRAPE systems. pages 41–48. John Wiley & Sons Ltd, 1998.

[17] M.D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004.

[18] O. Mencer. PAM-Blox II: Design and evaluation of C++ module generation for computing with FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 67–76, 2002.

[19] T. Narumi, R. Susukita, T. Ebisuzaki, G. McNiven, and B. Elmegreen. Molecular dynamics machine: Special-purpose computer for molecular dynamics simulations. In *Molecular Simulation*, pages 401–415, 1999.

[20] S. K. Okumura, J. Makino, T. Ebisuzaki, T. Fukushige, T. Ito, D. Sugimoto, E. Hashimoto, K. Tomida, and N. Miyakawa. Highly parallelized special-purpose computer, grape-3. In *Field Programmable Logic and Applications*, volume 45, pages 329–338, 1993.

[21] C. Patterson. High Performance DES Encryption in Virtex FPGAs using JBits. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 113–121, 2000.

[22] J. Stine and M. Schulte. The symmetric table addition method for accurate function approximation. In *Journal of VLSI Signal Processing*, pages 167–177, 1999.

[23] H. Touati and M. Shand. PamDC: a C++ library for the simulation and generation of Xilinx FPGA designs. 1996. http://research.compaq.com/SRC/pamette/PamDC.pdf.

[24] S. Waser and M. J. Flynn. *Introduction to arithmetic for digital systems designers*. Holt, Rinehart and Winston, 1982.