Rapid Development of High Performance Floating-Point Pipelines for Scientific Simulation

G. Lienhart, A. Kugel and R. Männer
Dept. for Computer Science V, University of Mannheim, B6-26B, D-68131 Mannheim, Germany
{lienhart,kugel,maenner}@ti.uni-mannheim.de

Abstract

In the last years, FPGAs became capable of performing complex floating-point based calculations. For many applications, highly parallel calculation units can be implemented which deliver a better performance than general-purpose processors. This paper focuses on applications where the calculations can be done in a pipeline, as it is often the case for simulations. A framework for rapid design of such calculation pipelines is described. The central part is a Perl based code generator, which automatically assembles floating-point operators into synthesizable hardware description code where the generator is directed by a pipeline description file. The framework is supplemented by various floating-point operators and support modules, which allow generating ready-to-use pipelines. The code generator dramatically reduces development time and produces high-quality results. The performance of the framework is demonstrated by the implementation of pipelines for gravitational forces and hydrodynamics.

1. Introduction

With the progress of computer technology, computational investigation of physical models by simulation became a research field of major importance. Due to its high computational demand special purpose computers found particular interest.

With current FPGA technology, it became possible to design custom computing platforms without the need of constructing ASICs. Even more important for scientific applications is the option to modify the implementation in the field as state-of-the-art algorithms are often subject to change. As the programmable logic resources of today's FPGAs allows the design of fast and highly complex computation units, FPGA based reconfigurable computing attracts more and more attention in the scientific and engineering community. Already published implementations of this kind are for example [9] were an FPGA based coprocessor for astrophysics simulations has been presented, [2] where the realization of a particle based hydrodynamics simulation with FPGAs is shown, and [7] where a force calculation pipeline for molecular dynamics is discussed.

Simulations of physical models usually require arithmetic with high precision and a wide dynamic range. For that, floating-point arithmetic is commonly used. This paper focuses on applications where the calculations can be arranged in a pipeline. If there is enough space on the target FPGA to implement one calculation unit for each operation then a direct pipeline approach leads to the fastest possible implementation. However, high-performance designs do not only need high-speed operators but also require that the architecture is carefully assembled. To find the solution, which provides the best synthesis results, several iterations may be needed. As physical simulations may easily require 50 or more operations inside the pipeline, the manual design of such pipelines is a tedious and error-prone task. Our framework automates most of this work and lets the designer concentrate on the architecture of the calculation unit.

The outline of this paper is as follows. After reviewing related work, we concretize the demands concerning the generation of FPGA designs by discussing the structure of our target applications and our implementation of a floating-point library. We then describe our framework design and the pipeline code generator. We will demonstrate the performance by showing the implementation results for two highly complex pipelines that are used for astrophysical simulations. Our approach is not limited to the FPGA for which we present the results, but works for any modern FPGA (e.g., Xilinx Virtex4 is already supported).

2. Related Work

As current developments in FPGA technology makes the implementation of floating-point computing circuits more and more attractive there is an ever-increasing activity in this field. Therefore, much work has been done

This work is supported by the Volkswagen Foundation.

1-4244-0054-6/06/$20.00

on designing floating-point units on FPGAs and the following can only provide a fragmentary overview. In [1] the authors presented a parameterized library of floating-point operators, which supports adders and multipliers up to single precision. In [2] a parameterized library for operands up to single precision was described, which additionally contains divider and square root units. In [3] the authors pointed out the benefits from hardware multipliers and shift registers for the design of floating-point adders, multipliers and dividers on modern FPGAs. In [4] the effects of pipeline depth of floating-point multipliers and adders for operators up to double precision were discussed. For comparison with commercial libraries, see e.g. [5] and [6].

Regarding tools for rapid development of FPGA designs from high-level description, most work has been done in the area of digital signal processing. In [12] the authors introduce JHDL, a Java based circuit design environment. In [13] an environment is described where MATLAB code is automatically translated to C code for CPUs and VHDL sources for the reconfigurable computing part of an algorithm. Well established approaches, which use C code with some language extensions at design entry, are Handel-C and System-C [14]. All these languages have in common that they aim at producing FPGA designs for a wide range of applications. This means that the designer must have a very clear view, how the compilers translate codes to FPGA designs in order to produce high-quality results. Further, utilizing parameterized external IP cores for arithmetic operations, if at all possible, means to write non-intuitive code. In contrast to this, there are block-based design approaches like the Xilinx system generator for the Mathworks Simulink interface [15]. Such tools allow a graphical design entry and simulation, and automated implementation of FPGA designs. However, introducing new parameterized calculation units means a high effort. Besides, graphical entry of pipelines with some 10 operators is much less intuitive than writing down the formulas like in a programming language. In [11] the authors presented a Perl based framework with the aim of generating VHDL code for calculation units by compiling object oriented Perl code. Floating-point numbers with parameterized precision were represented by objects and the computation structure was set by connecting these objects with overloaded operators. The framework has a built-in optimization algorithm for finding the best parameter set for operand precision. In [9] a code generator for many-body simulations is presented which creates both FPGA design code and software. The tool supports floating-point based pipelines but is limited to a predefined computation pattern.

In our work, we aim at providing a widely applicable design environment, which supports many different and specialized parameterized floating-point operators but has an easy-to-use design entry and is easy to extend for future libraries. None of the mentioned development platforms fits with these requirements. We will present how our implementation of a design framework achieves our goals for generating high-quality pipeline calculation units.

3. Target Applications

We are mainly interested in simulations where a physical system is represented by particles and the development of the system over time shall be investigated. Applications of that kind can be found e.g. in astrophysics, molecular dynamics and engineering.

In such simulations, usually a time step scheme is applied and according to Newton's laws, the motion of the particles is driven by the forces on these particles. Force calculations commonly consume the main fraction of computing time. The force on a particle results from the accumulated contributions from the interaction partners. These may be all other particles or only a limited number of neighboring particles.

The abstract computing structure is like in many other simulation applications too. All data related to a particle with index k shall be labeled with the same index ($x_k^1 .. x_k^n$). Computing a physical value y_i for a particle with index i, where a subset of other particles contribute (indices $j \in N(i)$) demands a summation over a term that depends on variables with index i and j.

$$y_i = \sum_{j \in N(i)} F\left(x_i^1 .. x_i^n, x_j^1 .. x_j^n\right) \qquad (1)$$

This structure of computation leads to the hardware scheme shown in Figure 1, where the data x_k^1 is stored in memory.

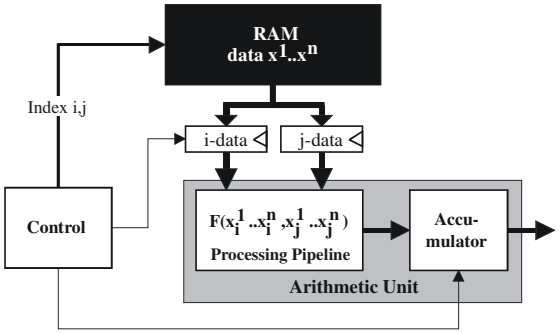

Figure 1. Hardware scheme for abstract computation pattern.

A well-known example is the treatment of gravitational interactions in astrophysics simulations:

$$\vec{F}_i^{grav} = -G\, m_i \sum_{i \neq j} \frac{m_j(\vec{r}_i - \vec{r}_j)}{\left(\varepsilon^2 + \left|\vec{r}_i - \vec{r}_j\right|^2\right)^{\frac{3}{2}}} \qquad (2)$$

The constant ε is an artificial smoothing factor that avoids unphysical high forces for close encounters of particles (in typical simulations a single particle may represent thousands of stars). This problem has exactly the structure of equation (1). The main part developing an FPGA design for that is to assemble the inner sum as a pipeline. In the comparatively simple case of equation (2) the pipeline can be composed using 6 adders, 9 multipliers, 1 square root unit and 1 divider.

4. Floating-Point Library

To design pipelines for scientific calculations we implemented a floating-point library with a full set of basic operators. Besides being parameterized in precision, the library was developed with the aim to benefit as much as possible from the algorithm with respect to economical use of logic resources. Therefore, many special operators for different constraints concerning input range and output quality were designed. Different operators are used for the addition of signed and unsigned values. A square operator instead of a multiplier is used when a square operation is performed. There are different operators for different rounding modes (truncation, round to nearest even, round to nearest number). Area and speed optimized versions of the operators are available. Even operators for special cases like mean value, multiplication by 3 or multiplication by a power of two have been designed, as well as special operators for vectors like cross product or norm of vector. Altogether, there are more than 30 different operators which must be supported by the pipeline generation.

To demonstrate the value of specializing operators concerning the savings of logic resources some performance numbers for different versions of adders and multipliers are collected in Table 1 and 2. For comparison, only data for single precision operators are shown. It shall be mentioned, that the IEEE 754 compliant precision of our operators may vary between 4 and 28 bit mantissa width. Our library is not limited to Virtex2 FPGAs but also supports the most recent Xilinx FPGAs as well as FPGAs from other manufacturers.

In Table 1 four versions of floating-point adders of our library and two commercial implementations are compared (data sheets of [5] and [6] from 2002 with implementations for Virtex2-4 FPGAs). The numbers differ quite remarkably. If speed is less important, e.g. if overall speed is anyway dominated by external memory, 15-20 % of the slices can be saved by going from the speed optimized version to the area optimized one. When there are only unsigned inputs then around 40 % of the slices can be saved without any loss of performance and even gaining a smaller latency.

Table 2 demonstrates the advantages from diversification of multiplier units comparing four units of our library and two commercial implementations. Squaring requires about 40 % less slices and saves one block multiplier compared to multiplying. Another 40 % of the slices can be saved by using the area optimized designs.

It shall be noted that by using *round to nearest number* mode (which is not IEEE 754 compliant) instead of *round to nearest even*, as for many algorithms that does not introduce significant errors, further 8-15 % of the slices for the adders and up to 15 % for the multipliers can be saved, depending on the mantissa width of the operators.

From these examples we see that the specialization of floating-point operators is essential for producing calculation pipelines that provide the best possible performance for a given problem. However, the variety of modules makes manual design of complex arithmetic units an even more demanding and error prone task. Our design methodology completely relieves the HDL developer from caring about module interfaces and operator latencies and therefore allows extremely short development cycles with ready for production results.

Table 1. Performance of different floating-point adders with single precision on Virtex2-4 FPGA[2].

Operator	Virtex 2 Slices	Speed (MHz)	Lat. (cyc.)
Unsigned Add, area opt.	165	81	4
Add, area opt.	276	78	6
Unsigned Add, speed opt.	208	143	6
Add, speed opt.	330	144	12
Nallatech Add	290	152	14
Quixilica Add	365	137	10

Table 2. Performance of different floating-point multipliers with single precision on Virtex2-4 FPGA.

Operator	Virtex 2 Slices	Block Mult.	Speed (MHz)	Lat. (cyc.)
Square, area opt.	53	3	97	4
Mult, area opt.	97	4	70	4
Square, speed opt.	89	3	143	5
Mult, speed opt.	149	4	138	5
Nallatech Mult	126	4	113	6
Quixilica Mult	358	0	128	7

5. Framework Design

The framework consists of two complementing developments. The first part is to embed the arithmetic units in a uniform interface structure in order to simplify the structural coding of pipelines as much as possible. It will

[2] All performance results were gained using a Xilinx Virtex2 FPGA XC2V6000-4, as we use this chip on our current prototype platform.

be described how this is done, so that the parameterization is completely preserved and the calculation units can easily be simulated. The second part is the code generator that automatically creates structural code for the pipeline architecture based on an architecture description file. This generator will be described in section 6 in detail. Figure 2 shows the overall structure of our framework. The gray boxes represent the elements, which are independent of the utilized floating-point core library and target platform. As can be seen, beside the floating-point library only some wrapper units need to be modified when new operators shall be supported.

In the current framework, all floating-point operators are grouped into five different classes of operators:

1. Unary operators
2. Binary operators
3. Operators with two input vectors and result vector
4. Operators with two input vectors and scalar output
5. Operators with one input vector and scalar output

The vector dimension is currently fixed to 3. All supported operators must comply with some design conditions. For each operator there must be a function which returns the latency according to parameters like mantissa width and pipeline depth. Beside signals for floating-point values, only a limited number of control signal inputs and outputs are supported (currently 4 in both directions). Any operator may have only one clock.

For enclosing operators into a uniform interface three elements of abstraction are used:

1. Enclose all signals which are related to a floating-point number into a record
2. Enclose all information related to the implementation of an operator into an operator descriptor record
3. Use a small number of wrapper components for instantiating the floating-point operators

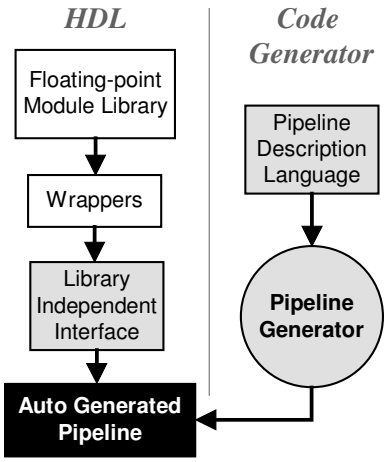

Figure 2. Framework for automatic pipeline generation.

In Figure 3 the hierarchical structure of the HDL part of the framework is drawn, which will be described in the following. The white boxes are the only ones which need to be modified when the floating-point library changes. The gray boxes exist in two versions, one for simulation and one for synthesis. The bold written types, functions and entities are used for assembling the pipelines.

The first element of abstraction, enclosing the floating-point number signals, is done via a VHDL record called *floPValType*. It consists of signals for mantissa, exponent and sign, an additional flag which shows if the number is equal to zero, and in the simulation library also a signal of type *real*. The latter takes the value of the represented number, which makes simulation much easier. The width of the mantissa and exponent is set to the maximum supported value, but usually only a fraction of it is used. In order to describe how many bits of the signals in *floPValType* are actually used, a record called *floPValDef* is created. There the size of the mantissa and exponent is stored, and the information if the sign and zero information are used. All types and support functions for dealing with the abstract floating-point representation are collected in the library part called *floPVal*. Regarding the descriptor record for floating-point operations, the following information is needed to completely describe any operator of our floating-point library:

- Type of operator
- Number of arguments and for each argument a descriptor record *floPValDef*
- Number of results and for each result a descriptor record *floPValDef*
- Pipeline depth
- Latency of operator
- Number of auxiliary input and output signals (e.g. reset, strobe, ready, error …)

All this information is stored in a record called *floPOpDescriptor*. The operator type is encoded with identifiers via an enumeration type *floPOpType*. Special features like non-standard rounding mode are defined by the type of the operator. Operator descriptors are generated via functions, one for each class of operators, according to the operator type and parameters. The latency information is calculated by calling the according function of the floating-point library (the presence of such a function was presupposed above).

All operators are enclosed inside two wrapper components, one for scalar operators and one for vector operators. There, the instantiation of an operator as well as its parameterization is selected by the *floPOpDescriptor* record, given as a *generic* in VHDL. The wrapper components do not use the *floPValType* records for floating-point ports (they are introduced in a second abstraction layer, see below). The operator type and descriptor records together with the wrapper components and operator

specific functions are collected in the library part called *enclosedFloPOp*. This part is the only one which changes when the floating-point library changes.

Therefore, the framework is very easy to extend. In order to add a new operator only the following little modifications must be done:

- Add a new identifier to *floPOpType*
- Extend function for generating *floPOpDescriptor*
- Extend wrapper component

Finally, a second abstraction layer is put over the *enclosedFloPOp* library part, called *abstractFloPOps*. There, basically floating-point component generators are implemented, one for each class of operators, which instantiate the wrapper component from *enclosedFloPOp*, but now use the abstract floating-point representation *floPValType*. Additionally there is a module for generating delay elements for floating-point numbers. The reason for introducing a second abstraction layer is that this part needs no changes when the floating-point library is extended. Therefore, this is a good place for including special simulation-only features by splitting the codes into a simulation and synthesis version. The simulation version takes care that for each floating-point signal a *real* value is kept up-to-date, which makes simulation of complex pipelines much easier.

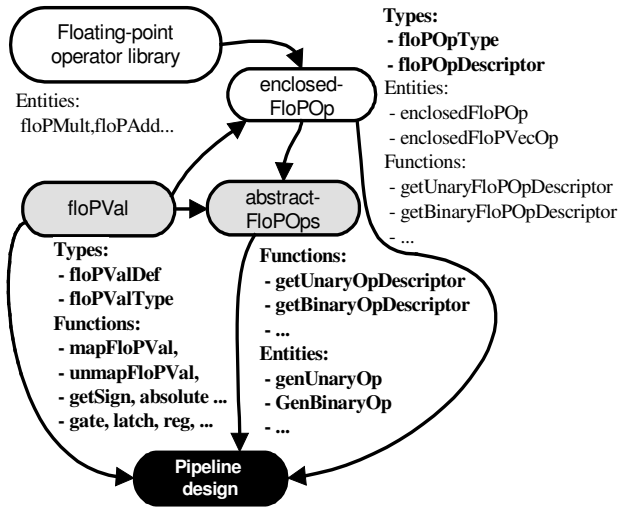

Figure 3. HDL framework for pipeline generator.

6. Pipeline Code Generator

With the HDL part of the framework as described in the last section, it is possible to assemble parameterized floating-point calculation pipelines with many different operators by using only a very small number of interfaces. However, it is still a time consuming and error prone task to create pipelines consisting of some 10 operators wherefore many hundreds of lines of HDL codes have to be written.

Our pipeline code generator completely takes on this part and only a pipeline description file in a simple to use language needs to be written. The generator is implemented in Perl. The decision to use Perl was made because of following qualities:

- Available on almost any platform
- Very capable built-in text manipulation features
- Very little language overhead allows compact code and fast development
- No need to care about variables for dynamic data structures

The VHDL code output of the generator was designed to benefit as much as possible from the capabilities of VHDL concerning calculation of parameters at the time of component instantiation and conditional elaboration of code. As the framework aims at assisting the FPGA designer with the HDL design flow, a key feature is, that the generated code is well readable and easy to simulate. The resulting generator implementation is very fast, robust and provides helpful error messages. Generating pipelines with 50 operations takes only a fraction of a second.

The language for describing the target pipeline architecture basically consists of three elements:

1. Define parameter sets for floating-point values
2. Declare variables
3. Set operations via expressions

Any floating-point parameterization is set via a descriptor *floPValDef* like in following example, where the mantissa and exponent width for a signed single-precision number without a zero-flag are defined:

```
floPValDef spDef(signifLength => 24,
                 expLength => 8,
                 useSign => 1, useIsZero => 0);
```

Both signals and floating-point variables can be declared. The type for floating-point values is called *floPVal* and any declaration of this type must contain an attached *floPValDef* descriptor:

```
FloPVal (a,b)(spDef);
```

Expressions for operations are written in the form of an equation. Signals can be treated with logical equations like in a hardware description language:

```
signal (a,b,c);
  a = b xor c;
```

Expressions for floating-point values are written with the `<...>` construct were the name of the operator is provided inside the angle brackets and the operator class is chosen by the number of operands on the left and right side as well as the number of results.

Examples are:

`z = <floPSquare> x;`	Unary operator
`z = x <floPAdd> y;`	Binary operator
`(rx,ry,rz)= (x,y,z)<floPVecDiff>(r,s,t);`	Operator with 2 vectors input and result vector
`d = (x,y,z)<floPVecDistance> (r,s,t)`	Operator with 2 vectors input and scalar output
`q = <floPVecSquare>(x,y,z)`	Operator with vector input and scalar output

Operator specific parameters, e.g. the pipeline depth, are set by a construct like follows:

```
<>.parameter_name = parameter_value;
```

The therewith-defined parameter is used for the preceding operator and all succeeding operators. There is also a custom module operator which allows to introduce modules which are not covered by the above described library part *abstractFloPOps*. The syntax is like follows:

```
<<entityName>(list_of_generics), list_of_ports>;
```

This construct can be applied to instantiate any VHDL component which has a *generic* declaration consisting of integers and *floPValDef* descriptors, and a list of ports with logic signals and *floPValType* signals.

A number of support modules are provided via functions like the following examples:

`s = getSign(x)`	signal s, return sign of floPVal x
`l = abs(x)`	return absolute value of x
`z = latch(x,s)`	save x when s is high
`q = gated(x,s)`	return x when s is low, else 0

Figure 4 shows an example for a pipeline description file which can be processed by the pipeline generator. The pipeline implements the calculation of the electrostatic potential from particles within a sphere of radius *rmax*:

$$\frac{1}{k} V(\vec{x}_i) = \sum_{j:\left|\vec{x}_i - \vec{x}_j\right| < rmax} \frac{q_j}{\left|\vec{x}_i - \vec{x}_j\right|} \qquad (3)$$

Figure 5 shows the architecture of the resulting design. The delay elements have been introduced automatically according to the latencies of the operators, and the control signal flow is generated correctly. The operator *unsignedFloPValGE* was implemented via the custom module operator, as it is not part of the floating-point library interface. This operator requires the presence of a package *floPValCmpPack*, which has been declared by the *package* statement. The name of the resulting pipeline module is provided via the *entity* statement and the default clock is set by the *clock* statement.

All signals, which are not assigned, are automatically treated as inputs to the module. Signals, which are not used at the right side of any expression, are treated as outputs. Type descriptors which are used for inputs or outputs are taken into the *generic* part of the resulting entity definition. Any other parameters can be put on the *generic* part when the specifier **`generic`** is used like with pipelineDepth in the example. The generator allows implicit declaration of variables when the type can be derived from the context. Signals may be declared **`static`**, which means that they do not need to be synchronized (like the signals *reset* and *maxr* in the example).

```
entity electrostaticPotential;

package floPLib.floPValCmpPack;

clock clk;

# parameters
floPValDef fpDef(signifLength => 24,
  expLength => 8,useSign => 1,useIsZero => 0);

generic int pipelineDepth = 1;

# inputs
signal (validIn_i,validIn_j,lastIn_j);
static signal reset;
floPVal (xIn,yIn,zIn,qIn)(fpDef);
static floPVal maxr(fpDef);

# outputs
floPVal potOut(fpDef);
signal validOut;

# save i-position
ix = latch(xIn,validIn_i);
iy = latch(yIn,validIn_i);
iz = latch(zIn,validIn_i);

# calculate distance
rij =(ix,iy,iz)<floPVecDistance>(xIn,yIn,zIn);
<>.pipelineDepth = pipelineDepth;

# suppress when rij > maxr
<<unsignedFloPValGE>(argDef=>fpDef),
 in argA(argDef)=>maxr, in argB(argDef)=>rij,
 out res(argDef)=>open, out A_ge_B=>suppress
>;
<>.latencyFunction = unsignedFloPValGELatency;

#calculate contribution to potential
pij = qIn <floPDiv> rij;
ppij = gated(pij, suppress);

# generate 'isFirst' strobe
isValid = validIn_i or validIn_j;
savedStrobe_i = latch(validIn_i,isValid);
isFirst = savedStrobe_i and validIn_j;

#accumulate
(potOut,validOut) = accu(ppij,validIn_j,
                         isFirst,lastIn_j,reset);
```

Figure 4. Example for pipeline description file.

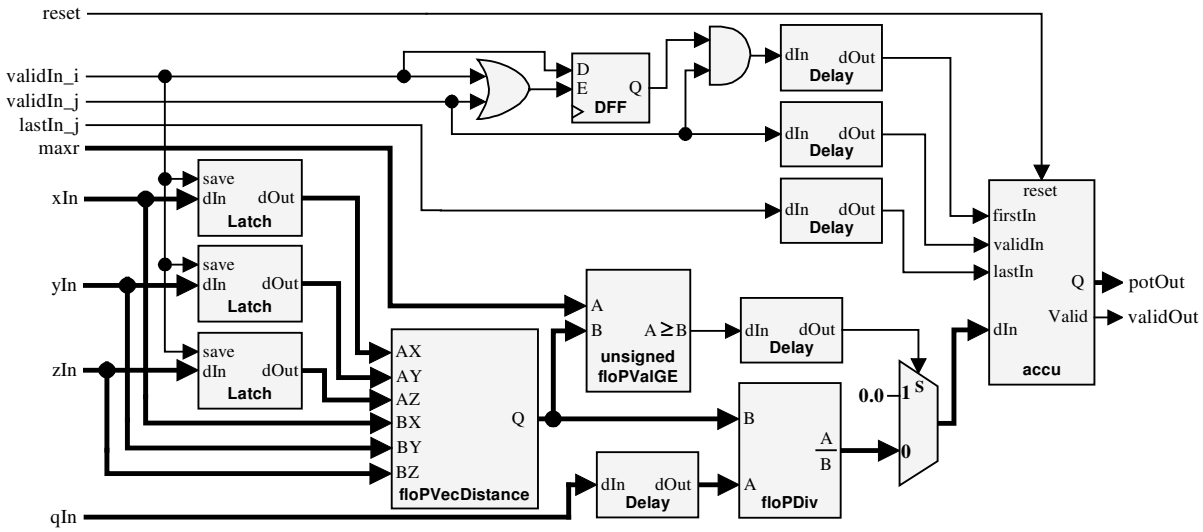

Figure 5. Synthesis result of example code from Figure 4.

The following is the resulting entity header:

```
entity electrostaticPotential is
generic(
 pipelineDepth: integer := 1;
 fpDef: floPValDef := init(signifLength=>24,
  expLength=>8, useSign=>true, useIsZero=>false)
);
port(
 clk: in std_logic;
 validIn_i: in std_logic;
 validIn_j: in std_logic;
 lastIn_j: in std_logic;
 reset: in std_logic;
 xIn: in floPValType;
 yIn: in floPValType;
 zIn: in floPValType;
 qIn: in floPValType;
 maxr: in floPValType;
 potOut: out floPValType;
 validOut: out std_logic
);
end electrostaticPotential;
```

The generated component is still fully parameterized. Additionally to the entity, a function called **`getElectrostaticPotentialLatency`** is produced for calculating the total latency according to the parameters. This makes the integration into the top-level designs very easy and moreover enables the hierarchical design of very complex parameterized calculation pipelines.

7. Results

We demonstrate the capability of our approach with two implementations of calculation pipelines. The first is related to gravity calculation for astrophysical N-body simulation. A pipeline has been generated which implements equation (2) and additionally the potential and the time derivative of the force as these are required for state of the art simulation codes. The resulting pipeline consists of 21 adders, four of them for unsigned inputs, 16 multipliers and 3 square units, one divider, one square root and one multiply-by-3 element, in total 43 operators. The second example is the calculation unit for hydrodynamic forces by the smoothed particle hydrodynamics algorithm (see e.g. [16]). This is a standard method to deal with gaseous matter in astrophysics simulations. The pipeline consists of 65 operators and special operators are extensively used. Operators with 16 mantissa bits precision were used, as this is sufficient for the target application.

Table 3 shows the implementation results for both pipelines comparing different settings.

Table 3. Implementation results for different pipelines. The number of adds #A, unsigned adds #U, mults #M, squares #S, square roots and divisions #R and special operators #X are shown (one operator of category X may merge several operations). The area and speed results for a Virtex2-4 FPGA and the number lines in the pipeline description file and VHDL code are shown (#PPL/#VHDL).

Pipeline	#A #U	#M #S	#R #X	Tot. #Ops	Slices BM	Freq. /MHz	#PPL #VHDL
Grav.	17	16	2	43	9872	71	237
(area)	4	3	1		73		3630
Grav.	17	16	2	43	12527	102	271
(speed)	4	3	1		73		4558
Grav.	21	19	2	43	15412	99	264
Xilinx	-	-	1		76		4255
SPH	8	17	5	65	7170	79	-
(manual)	7	8	8		25		3302
SPH	8	17	5	65	6972	82	188
(auto)	7	8	8		25		3137

The gravitation calculation pipeline was implemented with our framework using area and speed optimized floating-point units of our own library and the speed-optimized modules from the new XILINX core library (but still using the multiply-by-3 element). Using our library led to a saving of 19 % slices and 3 block multipliers at equal performance. Going to the area-optimized designs even a saving of 36 % resulted.

The results for the SPH pipeline contrasts a manually generated pipeline and the corresponding automatically generated one, using the same operators in both cases. While achieving the same speed, the automatic process led to an even more efficient design concerning the number of slices due to more efficient use of delay elements, although the difference is not dramatic. But the saving in development time (about 10x), measured by a saving in written code lines of about 15x is quite impressive.

8. Conclusions

We presented a methodology for the design of floating-point based calculation pipelines. We introduced a HDL framework design for embedding parameterized floating-point modules in a uniform abstract interface, independent of the actually used calculation cores. It enables systematically building compact code, which is easy to simulate.

Based on this abstraction layer a tool for automatically generating VHDL code for arbitrary complex calculation pipelines was presented. By designing an architecture in a very intuitive programming-like way with a specially designed pipeline description language, ready-to-use FPGA designs with production quality are generated. Even arbitrary control signal flow is supported which means that designs can be generated which fit without modification to any top-level control architecture. The design examples were presented for Virtex2 FPGAs but up-to-date FPGAs like Virtex4 are already supported.

Our approach frees the FPGA designer from the tedious and error-prone process of producing calculation pipelines. With this work, the design of highly complex calculation pipelines for scientific simulation applications can be done at the speed of writing software.

References

[1] P. Belanovic, M. Leeser, "A Library of Parameterized Floating-Point Modules and Their Use", *Proc. of the 12th International Conference on Field-Programmable Logic and Applications (FPL)*, 2002, pp. 657-666.

[2] G. Lienhart, A. Kugel, and R. Männer, "Using Floating-Point Arithmetic on FPGAs to Accelerate Scientific N-Body Simulations", *Proc. of the 10th International symposium on Field Programmable Custom Computing Machines*, 2002, pp. 182-191.

[3] E. Roesler, B.E. Nelson, "Novel Optimizations for Hardware Floating-Point Units in a Modern FPGA Architecture*", Proc. of the 12th International Conference on Field-Programmable Logic and Applications (FPL)*, 2002, pp. 637-646.

[4] G. Govindu, L. Zhuo, S. Choi and V. Prasanna, "Analysis of High-performance Floating-point Arithmetic on FPGAs", *Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[5] Nallatech Limited, "IEEE-754 compatible floating point cores for Virtex-II FPGAs", www.nallatech.com, 2002.

[6] Quixilica Limited, "Quixilica Floating Point FPGA Cores DataSheet", www.QinetiQ.com/quixilica, 2002.

[7] R. Scrofano and V.K. Prasanna, "Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware", *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2004.

[8] K.H. Tsoi, C.H. Ho, H.C. Yeung and P.H.W. Leong, "An Arithmetic Library and its Application to the N-body Problem", *Proc. FCCM'04*, 2004, pp. 68-78.

[9] T. Hamada and N. Nakasato, "PGR: A Software Package for Reconfigurable Super-Computing", *Proc. of the 15th International Conference on Field Programmable Logic and Applications (FPL)*, 2005, pp. 366-373.

[10] K. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance", *Proc. of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, 2004, pp. 171-180.

[11] C.H. Ho et al., "Rapid Prototyping of FPGA Based Floating Point DSP Systems", *IEEE International Workshop on Rapid System Prototyping 2002*, pp. 19-24.

[12] B.L. Hutchings, B.E. Nelson, "Using general-purpose programming languages for FPGA design", *Proc. of the 37th Conference on Design Automation*, 2000, pp. 561-566.

[13] P. Banerjee et al., "Overview of a Compiler for Synthesizing MATLAB Programs onto FPGAs*", IEEE transactions on VLSI*, vol. 12, no. 3, 2004, pp. 312-324.

[14] Celoxica Ltd, http://www.celoxica.com/technology/fpga/, 2005.

[15] "Xilinx System Generator v7.1 User Guide", Xilinx Inc. , 2005.

[16] W. Benz, “Smooth Particle Hydrodynamics: A Review”, J.R.Buchler(ed), *The Numerical Modelling of Nonlinear Stellar Pulsations*, J. R. Buchler (ed.), Kluwer Academic Publishers, 1990, pp. 269-288.