StReAm: Object-Oriented Programming of Stream Architectures using PAM-Blox

Oskar Mencer, Heiko Hübert, Martin Morf, Michael J. Flynn

Computer Systems Laboratory, Department of Electrical Engineering
Stanford, CA 94305, USA
email: {oskar,heiko,morf,flynn}@arith.stanford.edu
http://arith.stanford.edu/PAM-Blox/

Abstract

Simplifying the programming models is paramount to the success of reconfigurable computing. We apply the principles of object-oriented programming to the design of stream architectures for reconfigurable computing. The resulting tool, **StReAm**, is a *domain specific compiler* on top of the object-oriented module generation environment PAM-Blox. Combining module generation with a high-level programming tool in C++ gives the programmer the convenience to explore the flexibility of FPGAs on the arithmetic level and write the algorithms in the same language and environment.

Stream architectures consist of the pipelined dataflow graph mapped directly to hardware. Data streams through the implementation of the dataflow graph with only minimal control logic overhead. The main advantage of stream architectures is a clock-frequency equal to the data-rate leading to very low power consumption.

StReAm takes C++ expressions and converts them to a pipelined, scheduled stream architecture, including FIFO buffers for intermediate storage. The state of PAM-Blox hardware objects includes size and scheduling attributes of the object. The **StReAm** abstraction of PAM-Blox can handle combinational, fully pipelined, and sequential (iterative) arithmetic units.

We show a set of benchmarks from signal processing, encryption, image processing and 3D graphics in order to demonstrate the advantages of object-oriented programming of FPGAs.

1. Introduction

In this paper we present **StReAm**, a domain specific compiler for programming FPGAs. **StReAm** is build on top of the module generation environment PAM-Blox[10]. We start by reviewing some advantages of FPGAs for computation.

FPGAs offer reconfigurability on the bit-level at the cost of larger VLSI area and slower maximal clock frequency compared to custom VLSI. SRAM-based FPGAs allow fast reconfiguration of the entire chip. Thus, FPGAs are programmable devices that could compete with, or complement microprocessors.

Since their introduction, FPGAs have shown the potential for high performance (or throughput) and low power computation[28]. High performance and low power are a result of exploiting high degrees of parallelism and pipelining:

- **Parallelism**: FPGAs enable us exploit parallelism on the bit-level, arithmetic level, instruction level (ILP for microprocessors), and application level. FPGAs follow a long tradition of architectures that enable parallelism, such as massively parallel computing, superscalar and VLIW processors. Transforming the sequential description of an algorithm in order to compute more operations in parallel is usually called "extracting parallelism"–a process that can be as painful as it sounds if the algorithm is not cooperative.

- **Pipelining**: FPGAs have programmable registers in every cell making them natural candidates for highly pipelined architectures. For example, vector processors utilize pipelining of the data stream to achieve high throughputs. Systolic arrays[2][3] offer a regular structure that can be pipelined for high throughput applications. Stream architectures can be viewed as an extension of vector processing and systolic arrays. A pipelined dataflow graph is mapped directly into hardware and the data streams through the microarchitecture. Initial results show a high potential for stream architectures to improve performance and power consumption[28] by an order of magnitude over conventional microprocessors.

Simplifying the programming models is paramount to the success of reconfigurable computing. In order to simplify the programming of FPGAs it is necessary to hide a CAD tool within a compiler. In this paper we explore object-oriented programming of pipelined **StReAm** architectures for FPGAs.

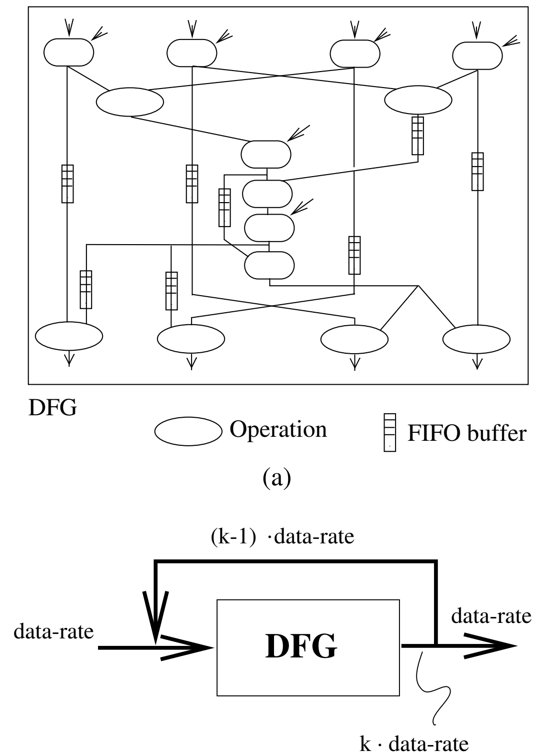

Figure 1: (a) shows an acyclic dataflow graph (DFG) with operations and distributed FIFO buffers. (b) shows a DFG with a loop($1 \ldots \mathtt{k}$), and the associated clock frequencies based on the data-rate.

1.1. Programming FPGAs–Related Work

FPGAs are programmable devices. Yet, programming FPGAs, or writing compilers for FPGAs, is much more complex than writing compilers for microprocessors. Because FPGAs offer programmability of the logic and the interconnect, compiling a program into an FPGA is similar to a VLSI CAD system. On the other hand the CAD design flow is not practical for programming FPGAs. The main questions are: How should the programming language express the various levels of parallelism available in the FPGA? How to express the timing of the design? How to explore area-time tradeoffs? How to debug the program? How to verify the compiler and module generator libraries?

HandelC[5] is a hardware (FPGA) programming language based on communicating sequential processes[4]. Every expression implies a latency of one clock cycle. Area-time tradeoffs can be explored by rewriting expressions.

JHDL[11] and Pebble[8] are examples for structural languages for FPGAs on the PAM-Blox level. JBITS[12] is an object-oriented environment on the FPGA configuration bitstream level.Neither JHDL nor Pebble offer operator overloading. Instead, expressions are constructed by nesting function calls.

Novel architectures combining a microprocessor with reconfigurable logic are natural targets for hardware compilation using software languages. For example, Callahan[17] compiles C for the Garp processor including a reconfigurable datapath.

Most available programming environments target general microarchitectures. **StReAm** is a domain specific compiler targeted at a specific architecture–stream architectures. The main objective is to limit the set of resulting architectures in order to achieve a very efficient implementation.

1.2. Stream Architectures

General purpose microprocessors are designed to deliver low latency computation with maximal clock frequencies leading to high power consumption. In terms of performance and power consumption, latency tolerant applications can be efficiently implemented with architectures that provide high throughputs, such as Imagine[23], Score[16], RaPiD[24], and the PCI-PipeRench[15].

Stream architectures[15] are fully pipelined, high throughput microarchitectures. Algorithms are executed by building the dataflow graph in hardware and streaming the data through the architecture. In the best case, when all the loops are fully unrolled, the dataflow graph is acyclic and the clock frequency of the design is the data-rate.

Figure 1 (a) shows the execution model of the stream architecture with completely unrolled loops. In a microprocessor the data goes from the register-file to the arithmetic unit. The result of the arithmetic operation can be forwarded to another arithmetic unit and is finally written back to the register file. In a stream architecture the role of the register file is taken over by distributed FIFO buffers with a delay equal to the time between production of the value and consumption of the value by the following operation. The distributed FIFO buffers enable us to exploit temporal data locality, while streaming data through pipelines exploits spatial data locality.

Unrolling a loop of a program requires the duplication of the loop-body in hardware. If complete unrolling of the loops requires more than the available hardware resources, we can not fully unroll the loops. In this case there are two options:

1. *Increasing clock frequency or reducing data-rate:* The resulting stream architecture has a loop. The clock-frequency is equal to the data-rate times the remaining number of loop iterations ($\mathtt{k}$) shown in figure 1 (b).

$$\mathtt{clockfrequency} = \mathtt{data\_rate} \cdot \mathtt{k} \quad (1)$$

2. *Reconfiguration:* We can partition the design into multiple configurations and stream the data through each

configuration separately.

In **StReAm** we focus on the first option, increasing the clock frequency when necessary. The **StReAm** extension to PAM-Blox[10] allows the programmer to design stream objects in C++.

2. PAM-Blox: Module Generation

Traditional VLSI design for high-performance ASICs consists of complete hand-layout of the data-path and high-level compilation of the control circuit. FPGAs do not offer the high flexibility of silicon area. For data-paths it is therefore sufficient to specify the logic, map the logic to lookup-tables and specify their location using Xilinx Netlist (XNF) directives.

PamDC[1] maps a structural description on the gate level to a Xilinx netlist. Experience with PamDC has shown that a low level, structural representation of FPGA circuits in C++ is very well suited for high-performance FPGA design. The major drawback of PamDC is the low level of design. In order to simplify the design process, we introduce additional levels of abstraction on top of PamDC. Figure 2 shows an overview of the PAM-Blox system including **StReAm**.

PamBlox is a template class library for hardware objects of low complexity, such as adders, counters, etc. *PaModules* are complex, fixed circuits implemented as C++ objects. PaModules may consist of multiple PamBlox and are optimized for a specific data-width. Examples are constant(k) coefficient multipliers (KCMs), Booth multipliers, dividers, and special purpose arithmetic units such as a constant multiply modulo $(2^{16}+1)$ operation for IDEA encryption[28].

PAM-Blox simplify the design of datapaths for FPGAs by implementing arithmetic module-generators in object-oriented C++. With PAM-Blox, hardware designers can benefit from all the advantages of object-oriented system design such as:

- *Inheritance:* Code-reuse is implemented by a C++ class hierarchy. Child objects inherit all public methods (function) and variables (state). For example, all objects with a carry-chain, such as adders, counters, and shifters, inherit the absolute and relative placement functions from their common parent.

- *Virtual Functions and Function Overloading:* Function overloading enables selective code-reuse. Part of the parent object can be redefined by overloading of inherited (virtual) methods. For example, a two's complement subtract unit can be derived from an adder by forcing a carry-in of one, and inverting one of the inputs.

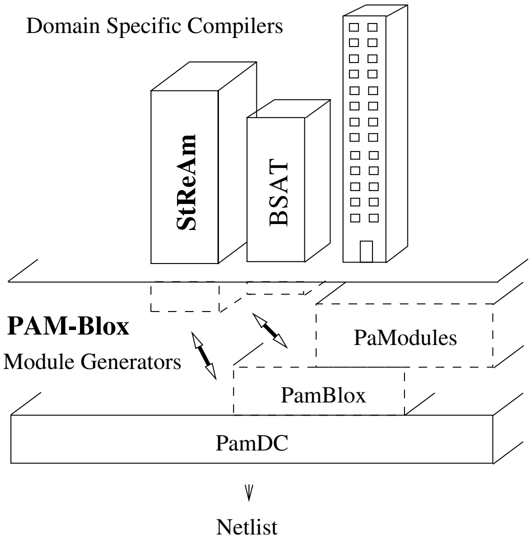

Figure 2: The figure shows the "city-model" for programming FPGAs. Vertical domain specific compilers such as **StReAm** and BSAT[31] sit on top of a horizontal foundation of PAM-Blox layers for module-generation.

- *Template Class:* The template class feature of C++ enables us to efficiently combine C++ objects and module-generation. In case of an adder, the template parameter is the bit-width of the adder. The instantiation of a particular object based on the template class creates an adder of the appropriate size.

- *Operator overloading and template functions* are used by **StReAm** described below.

3. StReAm: Programming FPGAs

StReAm is a domain specific tool build on top of PAM-Blox. Figure 2 shows the "city-model" for programming FPGAs. **StReAm** uses operator overloading and template functions in C++ to create dataflow graphs which are consecutively scheduled to obtain a stream architecture. The nodes of the stream architecture are mapped to PAM-Blox modules, to create a netlist file for the Xilinx place-and-route tools. Figure 3 shows the details of the class hierarchy of the system.

StreaModule is a subclass of PaModule. Applications are written as subclasses of StreaModule. **StReAm** enables high-level programming on the expression level. **StReAm** includes automatic scheduling of stream architectures, hierarchical wire naming and block placement. In addition, **StReAm** supports structural programming and allows the designer to combine structural descriptions with expressions. Thus, **StReAm** simplifies the design of complex stream architectures to just a few lines of code.

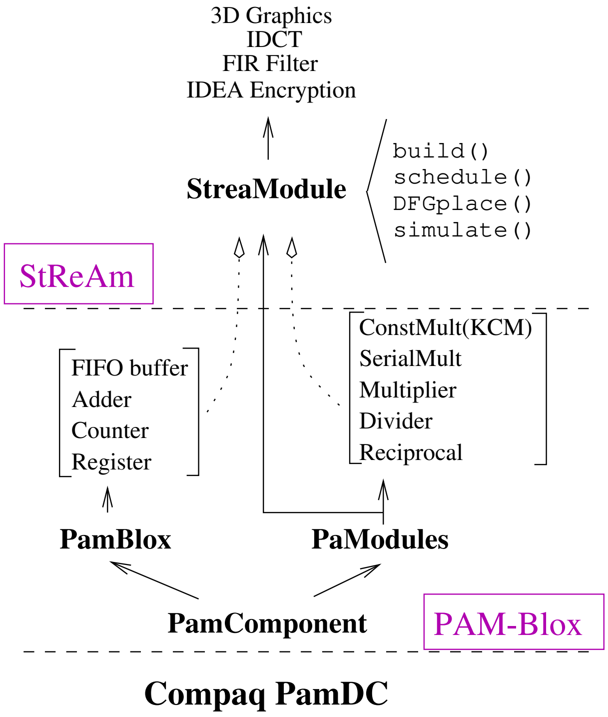

Figure 3: The figure shows **StReAm**'s class hierarchy on top of PAM-Blox. Arrows (→) denote a subclass, or a set of subclasses. The dotted and curved arrows denote operator overloading. Applications (stream architectures) are implemented as subclasses of StreaModules.

Designs are created by overloading the build() function of a StreaModule. A hardware integer (`HWint`) data type supports the common operators +,-,*,/,%. The user can define other operators and functions by utilizing operator overloading and template functions in C++. Extending the set of operators and functions requires manual design of optimized PamBlox or PaModules. Thus, the designer can adapt the arithmetic units to the specific needs of the application. The benchmarks in section 6 give a number of examples for application specific extensions and demonstrate the features mentioned above.

Calling the `<Name>::build()` function creates the dataflow required for automatic scheduling and placement. The `schedule()` function also creates all the required FIFO buffers and supplies the sequential and serial components with start signals. Finally, placement methods of hardware objects determine relative placement within the hardware objects.

StReAm currently supports arrays of the hardware integer type `HWint`, expressions with `HWint`'s and C++ integers resulting in hardware constants, and static '`for`' loops. In addition, more complex hardware units with possibly multiple inputs and outputs are implemented as template functions.

Currently, the resulting architectures are stream architectures with fully unrolled loops and no resource sharing in order to maximize throughput. Configuration registers can be added to the design in order to enable efficient communication between the host CPU and the reconfigurable device. An example using configuration registers is shown in section 6.4.

3.1. Families of Arithmetic Operators

One of the advantages of using FPGAs for computing is the flexibility on the arithmetic level. We define families of arithmetic units that are compatible with each other. As an example, the 4-bit digit-serial family consists of a 4-bit wide datapath with arithmetic units for addition, constant addition, and constant multiplication. The arithmetic family can be specified as a global parameter (used in the examples in section 6), or at the equation level.

StReAm supports the following arithmetic families:

- bit-serial
- 4-bit (nibble) serial
- parallel pipelined
- parallel combinational

The above arithmetic families are currently implemented for hardware integers. Future work includes extending the hardware types to other number representations such as logarithmic numbers (`HWlog`), fixed point numbers (`HWfix`), floating point numbers (`HWfloat`), the residue number system (`HWresidue`) based on the Chinese remainder theorem, redundant number representations, and rational number systems[32]. The main difficulty arises from the explosion of the number of necessary hand-designed arithmetic units. For example, in order to introduce `HWfix` numbers it is necessary to design arithmetic units for all operators and all possible combinations of inputs: (`HWint`,`HWfix`), (`int`,`HWfix`), and (`HWfix`,`HWfix`), taking into account the various formats that a fixed point number can take.

In addition to the basic arithmetic operators described above, PaModules include higher-level arithmetic modules such as CORDIC (Coordinate Rotation Digital Integrated Computer) units. For more information on the state-of-the-art in high-level arithmetic modules see [29].

StReAm gives the programmer access to these special arithmetic modules. Ideally, future special purpose high-level tools[30] will be able to map algorithms directly to such high-level arithmetic modules.

3.2. Debugging by Simulation

Debugging is one the main bottlenecks in programming any device. PamDC enables PAM-Blox and **StReAm** to run the compiled C++ executable to generate a netlist, or to simulate the design. Simulation occurs on the register-transfer level(RTL). Therefore **StReAm** simulations can optimally use C++ compiler optimizations. In addition, since most algorithms are initially written in C or C++, the software version can be easily used to verify the functionality of the FPGA design.

4. Scheduling Stream Architectures

StReAm automatically schedules the arrival of input data for each arithmetic component. The scheduling algorithm creates FIFO buffers and component start signals (for sequential components). The scheduler also calculates overall latency and data-rate for a given stream architecture. Our initial implementation utilizes "as soon as possible" (ASAP) scheduling, which is optimized for minimum-latency and resource unconstraint problems [6] without resource sharing.

Operator overloading creates a dataflow graph with arithmetic units as nodes. The scheduling algorithm traverses the dataflow graph including the components, data dependencies and scheduling information such as latency and throughput of each component. The scheduler retrieves the scheduling information from the state of the hardware objects. Using the dataflow graph, **StReAm** builds a sequencing graph, which contains all the information required for scheduling. A sample sequencing graph is given in figure 4. The sequencing graph contains the operations as vertices including their latency and data rate values. The edges of the sequencing graph are interconnecting wires which carry a time stamp. The time stamp of a wire is the time when a valid signal on this wire is generated. Initially, only the time stamps of input signals are set to zero. The remaining internal time stamps are set by the scheduling algorithm.

The scheduling algorithm performs the following four tasks:

- *Determine time stamps:* For each global output, the scheduler recursively traverses the sequencing graph. For each component, the time stamp (t_s) of the component output is set according to:

  $\texttt{ouput } t_s = \max(\texttt{input } t_s) + \texttt{component\_latency}$

- *Global latency and data rate calculation:* The scheduler calculates overall latency and data rate of a StreaModule while traversing the sequencing graph. Global latency is determined by the sum of latencies in the longest path of the design which is equal to the maximal time stamp of global StreaModule outputs. The data rate describes the time interval in which components consume and produce data. For sequential components the data rate is equal to the latency, whereas for pipelined components the data rate is usually smaller than the latency. The global data rate is determined by the minimal data rate of all components in the design.

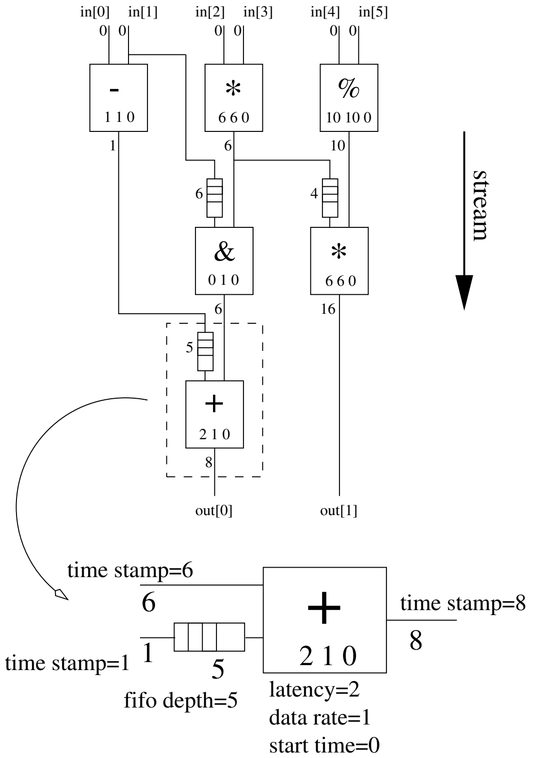

Figure 4: The figure shows the scheduled data-flow graph implementing the equations: $out[0] = (in[0] - in[1]) + in[1]\&(in[2] * in[3])$; $out[1] = in[2] * in[3] * (in[4]\%in[5]$. Sequencing information is given in the nodes (component latency, data-rate and start time) and interconnections include time stamps.

- *Introduction and configuration of FIFO buffers:* The scheduler ensures equal arrival times of component inputs by inserting delays. Delays are realized as distributed FIFO buffers. The dual-ported RAMs and registers available in a Configurable Logic Block (CLB) can be utilized to implement the FIFO buffer component. The FIFO depth ($d(i)$) for a given component input i is calculated as:

$$d(i) = \max(\texttt{input } t_s) - t_s(i) \qquad (2)$$

  The scheduler connects the dual-ported RAMs to a global address generator according to their FIFO buffer depth.

- *Supplying components with start signals:* Sequential and serial components require a start signal each time new input data can be applied to the component. Start time ($t_{\mathtt{start}}$) depends on the maximum input time stamp (t_s) and the previously calculated global data rate.

$$t_{\mathtt{start}} = \max(\mathtt{input}\ t_s) \bmod \mathtt{data\_rate}^{-1} \quad (3)$$

A global state counter generates the required start signals.

4.1. Dealing with Precision and Overflow

Each arithmetic unit (PamComponent) includes a precision value and an overflow bit as part of the state of the hardware object. Hardware objects representing the arithmetic units are the nodes of the dataflow graph. The precision value inside the hardware object can be used at compile time to evaluate error propagation through the dataflow graph.

The stream architecture includes overflow detection per data item. The overflow bit is part of the `HWint` type. The overflow bit of the output of an arithmetic unit is set if the previous arithmetic operation overflows, or if any overflow bit of the inputs to the previous operation is set. Finally, each datum leaving the stream architecture has an overflow bit signaling the correctness of the result.

5. High-Level Object Placement

StReAm includes the option to explore high-level placement. Placement is done via congestion modeling developed for high-level placement of VLSI circuits [21]. On the object level, we have a set of blocks and a set of interconnections. The placer arranges the blocks to minimize a cost function, using simulated annealing. The placement is optimized with a non-slicing structure using the sequence-pair representation proposed by Murata [20].

In our case, the main objective is to maximize performance and minimize clock cycle time. Thus, we minimize area, wire length and/or wire congestion according to the following cost function:

$$\mathtt{cost} = \mathtt{area} + \lambda_1 \cdot \mathtt{wireLength} + \lambda_2 \cdot \mathtt{wireCongestion}$$

The details of the model are described in [21]. **StReAm** allows the designer to specify λ_1 and λ_2 in order to explore the solution space for optimal area and performance. Optimizing the algorithm for FPGA placement is work in progress.

6. Benchmarks and Results

We chose the following benchmarks from signal processing, encryption, image processing and graphics to demonstrate the advantages of designing stream architectures with **StReAm**. Final results show the performance of the final circuits for the Xilinx XC4000 family after Xilinx place and route tools.

6.1. FIR Filter

The following code creates FIR filters with constant coefficients. Operators '+' and '*' are overloaded to create the appropriate arithmetic units. Multiplying by a constant integer instantiates efficient constant-coefficient multipliers. Data width and datapath width are specified separately to enable digit-serial arithmetic. In the case below we implement a 16-bit FIR filter with 4-bit digit serial arithmetic units. The `delay` operator inserts the FIR filter delays (deltas) similar to the way delays are specified in the Silage language[6]. The variables `in[], out[]` are the inputs and outputs of the stream architecture, defined by setting `NUM_BLOCK_INPUTS,NUM_BLOCK_OUTPUTS`:

```
const int NUM_BLOCK_INPUTS=1;
const int NUM_BLOCK_OUTPUTS=1;
const int BITS = 16;
const int COMP_MODE = DIGIT_SERIAL;

const int STAGES=4;
const int coef[STAGES]={23,45,67,89};

HWint<BITS> delayOut;
HWint<BITS> adderOut;

void Filter::build()
{
  delayOut=in[0];
  adderOut=delayOut*coef[0];

  for (i=1;i<STAGES;i++){
    delayOut=delay(delayOut,1);
    adderOut=adderOut+delayOut*coef[i];
  }
  out[0]=adderOut;
}
```

Table 1 shows the results for four variations of the 4-tap FIR filter. The results show a 4-stage FIR filter implemented with combinational arithmetic units, and three pipelined versions. As expected the bit-serial design takes the smallest area with the longest latency. The parallel, pipelined version has higher throughput but requires most area. The lower part of the table shows the maximal number of stages that **StReAm** can fit on a Xilinx XC4020 FPGA with 800 CLBs. All designs are created with the same few lines of code shown above by simply setting the compiler parameter `COMP_MODE`.

Table 1: FIR Filter <preliminary> Results

	combi-national	pipelined parallel	digit-serial	bit-serial
4 stage FIR				
Area[CLB]	246	293	210	184
Cycle Time(CT)	70.1ns	20.8ns	21.2ns	24.2ns
Latency	3	9	17	52
Throughput	16bits/CT	16bits/CT	4bits/CT	1bit/CT
FIR Stages	6	6	14	17
Area[CLB]	332	432	678	635
Latency	5	11	57	260
Cycle Time(CT)	88.7ns	25.1ns	27.3ns	28.0 ns
Throughput	16bits/CT	16bits/CT	4bits/CT	1bit/CT

6.2. IDEA Encryption

IDEA (International Data Encryption Algorithm)[18] was developed by Xuejia Lai and James Massey. IDEA is a strong encryption algorithm developed for DSP microprocessors. IDEA encrypts or decrypts 64-bit data blocks, using symmetric 128-bit keys. The 128-bit keys are expanded further to 52 sub-keys, 16 bits each. The kernel loop (or round) is generally executed 8 times for either encryption or decryption. Hand crafted results for a stream architecture implementation of IDEA are presented in [28].

One IDEA round can be mapped to a four-input, four-output stream architecture. In order to fit two loops onto one Xilinx XC4020E FPGA we use digit serial arithmetic with a datapath width of 4 bits. The following code shows the four-input, four-output `IDEA::build` implementation of one round of IDEA encryption:

```
const int NUM_BLOCK_INPUTS=4;
const int NUM_BLOCK_OUTPUTS=4;
const int BITS = 16;
const int COMP_MODE=DIGIT_SERIAL;

// Encryption Key:
const int key[10]={9277,98,237,4,978,
                   122,723,3654,24,1536};
HWint<BITS> t[9];
HWint<BITS> temp;

void IDEA::build(){

  t[1] = ideaKCM16(in[0] , key[0]);
  t[2] = (in[1] + key[1]);
  t[3] = (in[2] + key[2]);
  t[4] = ideaKCM16(in[3] , key[3]);
  tmp = t[1] ^ t[3];
  tmp = ideaKCM16(tmp , key[4]);
  t[7] = (tmp + (t[2] ^ t[4]));
  t[8] = ideaKCM16(t[7] , key[5]);
  tmp = (t[8] + tmp);
  out[0] = t[1] ^ t[8];
  out[3] = t[4] ^ tmp;
  tmp = tmp ^ t[2];
  out[1] = t[3] ^ t[8];
  out[2] = tmp;
}
```

The resulting stream architecture with 14 arithmetic units and 8 automatically generated and scheduled FIFO buffers is shown in figure 1. In addition to operator overloading, IDEA requires a special `mod` $2^{16}+1$ constant multiplier implemented as a PaModule with fixed bitwidth, which is instantiated by the function `ideaKCM16()`. Final results are shown in Table 2.

6.3. Inverse Discrete Cosine Transform (IDCT)

The IDCT is used in signal and image processing (e.g. MPEG, H.263 standards). We implement an 8x8 1-dimensional IDCT. The StreaModule below is based on an optimized IDCT implementation [19].

```
const int NUM_BLOCK_INPUTS=8;
const int NUM_BLOCK_OUTPUTS=8;
const int BITS = 14;
const int COMP_MODE=PARALLEL;

const int coef[8] = {16069, 15137, 13623,
          11585, 11585, 3196, 6270, 9102};

HWint<BITS> t[18];

IDCT::build() {

t[0] =((in[0]+in[4])*coef[4]+256)>>9;
t[1] =((in[0]-in[4])*coef[4]+256)>>9;
t[2] =(in[2]*coef[6]-in[6]*coef[1]+256)>>9;
t[3] =(in[2]*coef[1]+in[6]*coef[6]+256)>>9;
t[4] =(in[1]*coef[0]+in[7]*coef[5]+256)>>9;
t[5] =(in[1]*coef[5]-in[7]*coef[0]+256)>>9;
t[6] =(in[3]*coef[2]+in[5]*coef[7]+256)>>9;
t[7] =(in[3]*coef[7]-in[5]*coef[2]+256)>>9;

t[8] =(t[0]+t[3]+2)>>2;
t[9] =(t[1]+t[2]+2)>>2;
t[10]=(t[1]-t[2]+2)>>2;
t[11]=(t[0]-t[3]+2)>>2;

t[0]=(t[4]+t[6]+2)>>2;
t[1]=(t[5]+t[7]+1)>>1;
t[2]=(t[4]-t[6]+1)>>1;
t[3]=(t[5]-t[7]+2)>>2;

t[4]=(((t[1]+t[2]+1)>>1)*coef[3]+8192)>>14;
t[5]=(((t[1]-t[2]+1)>>1)*coef[3]+8192)>>14;

 out[0]=(t[8]+t[0]+4)>>3;
```

```
 out[1]=(t[9]+t[4]+4)>>3;
 out[5]=(t[10]+t[5]+4)>>3;
 out[3]=(t[11]+t[3]+4)>>3;
 out[7]=(t[8]-t[0]+4)>>3;
 out[6]=(t[9]-t[4]+4)>>3;
 out[2]=(t[10]-t[5]+4)>>3;
 out[4]=(t[11]-t[3]+4)>>3;
}
```

The resulting stream architecture consists of 98 arithmetic units and 4 FIFO buffers. The final results are shown in Table 2.

6.4. 3D Motion: Real-time Translation and Rotation

In 3D graphics, a common problem is the translation and rotation of a large set of points in 3D. This stream of points is transformed by a translation vector and two 2D rotation angles obtained from one 3D rotation. The following implementation uses 2D CORDIC modules (`ROTATE()`) implemented as PaModules[30]. The **StReAm** program looks as follows:

```
const int NUM_BLOCK_INPUTS=3;
const int NUM_BLOCK_OUTPUTS=3;
const int BITS = 12;
const int COMP_MODE=PARALLEL;

HWint<BITS> x_in,y_in,z_in;    //inputs
HWint<BITS> x0,y0,z0,phi1,phi2;//rotation
HWint<BITS> dx,dy,dz;          //translation
HWint<BITS> x[2],y[2],z[2];    //temp coords

MOTION3D::build(){

 x_in = in[0];
 y_in = in[1];
 z_in = in[2];

 x0 = configReg[0];
 y0 = configReg[1];
 z0 = configReg[2];
 phi1 = configReg[3];
 phi2 = configReg[4];
 dx = configReg[5];
 dy = configReg[6];
 dz = configReg[7];

(x[0],y[0])=ROTATE((x_in-x0),(y_in-y0),phi1);
(y[1],z[1])=ROTATE(y[0],(z_in-z0),phi2);

 out[0]=x[0] + x0 + dx;
 out[1]=y[1] + y0 + dy;
 out[2]=z[1] + z0 + dz;
}
```

The StreaModule above takes 3 input coordinates (`x_in, y_in, z_in`). representing a point in space. The result is a rotated and translated point (`out[0...2]`). The center of rotation (`x0, y0, z0`), angles `phi1, phi2`) and translation vector (`dx, dy, dz`) are stored in configuration registers (`configReg`). The value of the configuration registers can be changed without reconfiguration of the FPGAs. Writing these eight configuration values to the FPGA configures the stream architecture to perform a particular 3D motion.

Table 2: Benchmark Results

	IDEA	IDCT	3D MOTION
Area[CLB]	460	463	320
Cycle Time(CT)	24.1ns	27.9ns	33.9ns
Throughput(bits/CT)	$4 \cdot (16/4) = 16$	12.4	36
Total Latency	17	15	27
Arithmetic	digit-serial 16-bit data	parallel* 14-bit	parallel 12-bit data

*sequential multiply

This example demonstrates a template function:`ROTATE`. C++ instantiates the `ROTATE` function to create CORDIC units[30] with the appropriate bitwidth based on the types of the input variables. In addition, the rotate function demonstrates a multi-input, multi-output module instantiation by overloading the "," operator in `(x[0],y[0])` leading to the efficient program above.

The code above results in 9 add/sub units, 2 CORDIC units and 1 FIFO buffer. The final results of a fully pipelined 3D rotation and translation module are shown in Table 2.

7. Conclusions and Future Work

StReAm applies an object-oriented design methodology to programming FPGAs. The advantages of object-oriented design with C++ which have been recognized in the software industry find their way into VLSI CAD[26][27] and into programming of FPGAs shown in this paper. FPGAs offer the flexibility to adapt the number representation, precision, and arithmetic algorithm to the particular needs of the application. Yet, in general it is difficult to explore completely different arithmetic solutions. Combining module generation with a high-level programming tool in C++ gives the programmer the convenience to explore the flexibility of FPGA on the arithmetic level and write the algorithms in the same language and environment.

For **StReAm** the key enabling C++ technologies are dynamic operator overloading and template functions. Furthermore, just as in PAM-Blox, the class hierarchy, inheritance, template classes and method overloading enable efficient code-reuse and code-management.

Future work needs to address the problem of debugging, simulation, and verification of module generators. Another important next step is to adapt the state-of-the-art in hardware/software co-design of parallel and pipelined systems[25] into a compiler for FPGAs or reconfigurable resources closely coupled with a microprocessor or memory. Ideally, such a compiler would be able to explore parallelism and pipelining on the algorithm level, instruction level, arithmetic level and bit level.

8. Acknowledgments

We would like to thank W. Luk, J. Wawrzynek, and Arvind for helpful discussions on module-generation and the programming approach to hardware design. Thanks to Compaq Systems Research Center for support of this work, and M. Shand for maintaining PamDC. Thanks to L. Séméria for discussions on the draft of this paper. The second author thanks his advisor Prof. H. Klar for support of this research.

9. References

[1] P. Bertin, D. Roncin, J. Vuillemin, *Programmable Active Memories: A Performance Assessment,* ACM FPGA, February 1992.

[2] H. T. Kung, *Why Systolic Arrays,* IEEE Computer, Jan. '82.

[3] H. M. Ahmed, J.-M. Delosme, M. Morf, *Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing,* IEEE Computer, vol. 15, no. 1, Jan. 1982.

[4] C.A.R. Hoare *Communicating Sequential Processes,* Prentice Hall International, London, 1985.

[5] Embedded Solutions *Handel C,* http://www.embeddedsol.com/

[6] G. DeMicheli *Synthesis and Optimization of Digital Circuits,* McGraw-Hill, 1994.

[7] W.H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. Prasanna, H. Spaanenburg, *Seeking Solutions in Configurable Computing,* IEEE Computer Magazine, December 1997.

[8] W. Luk, S. McKeever, *Pebble: A Language for Parametrised and Reconfigurable Hardware Design,* Field-Programmable Logic and Applications (FPL), Tallinn, Estonia, Aug. 1998.

[9] J.M. Emmert, A. Randhar, D. Bhatia, *Fast Floorplanning for FPGAs,* Field-Programmable Logic and Applications (FPL), Tallinn, Estonia, Aug. 1998.

[10] O. Mencer, M. Morf, M. J. Flynn, *PAM-Blox: High Performance FPGA Design for Adaptive Computing,* IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, 1998.
http://arithmetic.stanford.edu/PAM-Blox/

[11] P. Bellows, B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, M. Rytting, *A CAD Suite for High-Performance FPGA Design,* IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, 1999.

[12] S. A. Guccione, D. Levi and P. Sundararajan, *JBits: A Java-based Interface for Reconfigurable Computing,* 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD), Laurel, Maryland, 1999.

[13] M.B. Gokhale, J.M. Stone, *NAPA C: Compiling for a Hybrid RISC/FPGA Architecture,* IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, 1999.

[14] A. Koch, *Enabling Automatic Module Generation for FCCM Compilers,* Poster Session 1, IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, 1999.

[15] R. Laufer, R. Reed Taylor, H. Schmit *PCI-PipeRench and SWORDAPI: A System for Stream-based Reconfigurable Computing,* IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, 1999.

[16] Berkeley Brass Project, *SCORE: Stream Computations Organized for Reconfigurable Execution Fast Module Mapping and Placement for Datapaths in FPGAs,* http://www.cs.berkeley.edu/Research/Projects/brass/SCORE/

[17] T.J. Callahan, J. Wawrzynek, *Instruction-Level Parallelism for Reconfigurable Computing,* Field-Programmable Logic and Applications (FPL), Tallinn, Estonia, Aug-Sep 1998

[18] X. Lai, J.L. Massey, S. Murphy, *Markov Ciphers and Differential Cryptanalysis,* EUROCRYPT '91, Lecture Notes in Computer Science 547, Springer-Verlag, 1991.

[19] E. Linzer, E. Feig, *New Scaled DCT Algorithms for Fused Multiply/Add Architectures,* International Conference on Acoustics, Speech, and Signal Processing, Proceedings ICASSP .91, Vols.1-5, pp.2201-2204, 1991.

[20] H. Murata et al., *Rectangle-Packing-Based Module Placement,* Proc. of International Conference on Computer Aided Design, 1995.

[21] Patrick Hung and Michael Flynn, *Deep Submicron VLSI Floorplanning Algorithm,* Electronic Devices and Systems Conference, Nov. 1999.

[22] T.J. Callahan, P. Chong, A. DeHon, J. Wawrzynek, *Fast Module Mapping and Placement for Datapaths in FPGAs,* Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, Feb. 1998.

[23] S. Rixner, W.J. Dally, U.J. Kapasi, B. Khailany, A. Lopez-Lagunas, P.R. Mattson, J.D. Owens, *A Bandwidth-Efficient Architecture for Media Processing,* Proceedings of the 31st Annual International Symposium on Microarchitecture, Dallas, Texas, Nov. 1998.

[24] C. Ebeling, D.C. Cronquist, P. Franklin, J. Secosky, S.G. Berg, *Mapping Applications to the RaPiD Configurable Architecture,* IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, CA, 1997.

[25] S. Bakshi, D.D. Gajski, *Partitioning and Pipelining for Performance-Constrained Hardware/Software Systems,* IEEE Transaction on VLSI Systems, Dec. 1999.

[26] Technical Papers, *The SystemC Community,* http://www.systemc.org/

[27] S. Vernalde, P. Schaumont, I. Bolsens, *An Object Oriented Programming Approach for Hardware Design,* IEEE Workshop on VLSI'99, Orlando, April 1999.

[28] O. Mencer, M. Morf, M. Flynn, *Hardware Software Tri-Design of Encryption for Mobile Communication Units,* International Conference on Acoustics, Speech, and Signal Processing, Proceedings ICASSP, May 1998.

[29] J.M. Muller, *Elementary Functions, Algorithms and Implementation,* Birkhaeuser, Boston, 1997.

[30] O. Mencer, L. Séméria, M. Morf, J.M. Delosme, *Application of Reconfigurable CORDIC Architectures,* The Journal of VLSI Signal Processing, Special Issue: VLSI on Custom Computing Technology, Kluwer, March 2000.

[31] O. Mencer, M. Platzner, *Dynamic Circuit Generation for Boolean Satisfiability in an Object-Oriented Design Environment,* Hawaii International Conference on System Sciences (ConfigWare Track), Jan. 1999

[32] O. Mencer, *Rational Arithmetic Units in Computer Systems,* PhD Thesis (with M.J. Flynn), E.E. Dept., Stanford, Jan. 2000.