PAM-Blox II: Design and Evaluation of C++ Module Generation for Computing with FPGAs

Oskar Mencer

Computing Sciences Center, Bell Labs, Lucent,
Murray Hill, NJ 07974, USA.
email: mencer@research.bell-labs.com

Abstract

This paper explores the implications of integrating flexible module generation into a compiler for FPGAs. The objective is to improve the programmability of FPGAs, or in other words, the productivity of the FPGA programmer. We describe (1) the module generation library *PAM-Blox II*, the second generation of object-oriented module generators in C++, targeted at computing with FPGAs, and (2) examples of design tradeoffs and performance results using redundant representations for addition and multiplication, and technology mapping of comparison and elementary function evaluation.

PAM-Blox II is built on top of a set of extensions to the gate level FPGA design library PamDC to provide a more efficient, portable, scalable, and maintainable module generator library. Using PAM-Blox II we demonstrate a simplified interface to *bit-level programability*. The simplification results from the bottom-up approach and a close coupling of architecture generation, module generation and gate level CAD.

The tradeoffs for the module generators are based on trading area for speed and hand-optimizing technology mapping to the specific FPGA technology. As an example, we show that redundant number representations hold one key to unleashing the full potential of reconfigurability on the bit-level. The presented module generators are applied to encryption and compression to show the impact of the bit-level optimizations on application performance.

1. Introduction

The goal of computing with Field-Programmable Gate Arrays (FPGAs) as first investigated by the PAM[1] project and Splash[2] project, is to migrate certain computations that are inefficiently handled on microprocessors to FPGA accelerators. Field Programmable Gate Arrays (FPGAs) have a tremendous potential to increase performance[3] and reduce power consumption[4] for a wide variety of applications (e.g. [6][7][8][9]).

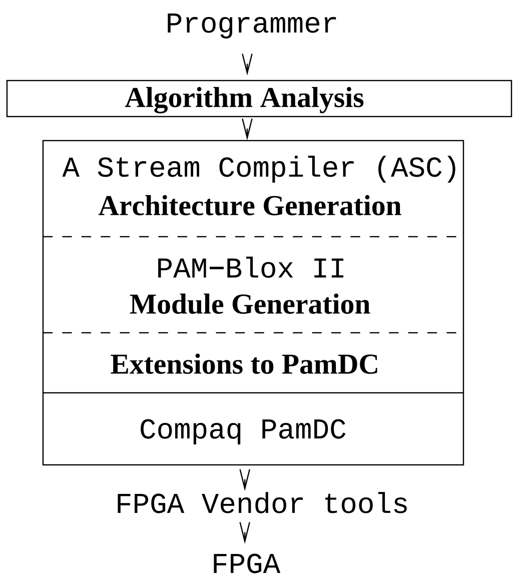

Figure 1: Partitioning the FPGA compilation problem into levels of abstraction.

We partition FPGA compilation into four tasks, or levels of abstraction as shown in Figure 1:

1. **Algorithm analysis layer:** Common tasks associated with this layer are: extracting compiler-controlled memory management[18][19], pointer analysis for hardware synthesis[17], loop transformation[10][11][12], precision analysis[13][14][15][16], data-structure transformations, and architecture selection. Some of these tasks could be handled automatically, while automating others is still beyond the state-of-the-art and has to be done manually.

2. **Architecture generation layer:** At this layer the representation contains variable-specific precision information, loop hints for generating the architecture, explicit memory allocation mapped to FPGA internal and external resources, and further hints to the software on how to generate the application specific in-

stance of the selected architecture. Usually, a compiler at this level focuses on one particular architecture such as a stream architecture[20] based on dataflow graphs, a parallel multi-processor-like (very coarse grain parallel) architecture[21], or more conventional parametrizable microprocessors[22][23]. These architectures are *domain specific architectures*, as there exists a domain of applications for which the specific architecture is the "optimal" choice.

3. **Module generation layer:** Module generator libraries define the "instruction set" for the architecture generation layer. The standard solution is to use FPGA vendor libraries which are very restrictive and can not easily be changed, adapted or extended due, partly, to intellectual property reasons. Since FPGAs have full flexibility at the bit level, module generation is at the core of achieving orders of magnitude speedup by mapping algorithms to FPGAs. We believe that the key requirements for module generation are: open source, scalability, extensibility, extensive documentation, efficient code sharing, maintainability, and features for portability to new generations of FPGAs. *This paper shows the second generation of a solution to module generation in object oriented C++.*

4. **Gate Level to Netlist layer:** This layer contains commercial CAD synthesis tools such as Synopsys FPGA Express[25], or mature research tools such as Compaq (DIGITAL) PamDC[26] used for the module generators in this work.

In order to meet the above requirements for module generation, we apply an object-oriented design methodology. Object oriented software design is a well established technology in the software world. The hardware world is slowly adopting the advances made by object oriented languages such as C/C++[27][28] and Java[29][30][31]. Object oriented design leads to an efficient solution of the module generation problem by focusing on the requirements for module generation mentioned above, such as scalability and code sharing. Inheritance and hierarchical class structures match the requirements of creating a large library of module generators with the logic expressed as computation (methods) and module abstraction parameters described as internal state (local variables) of the generated object.

Why object oriented C++? C++ is one of the richest object-oriented languages sometimes criticized for the complexity arising from this richness of features. On the other hand, once an optimal mapping of the problem space to C++ features is established, the software design and maintenance task is greatly simplified.

Why C++ as opposed to Java[29][30][31]? C++ offers operator overloading (not available in Java) which is one of the most convenient features for adding application specific semantics to a programming language. In our case these semantics include boolean logic equations, and in fact any expressions/operations on user-defined classes which specify hardware variable types. The second reason for using C++ is the Standard Template Library (STL)[32]. The STL offers various polymorphic data-structures and algorithms which are independent of the data type and as a consequence the user defined types (or module generator classes) can be used as the nodes of all STL data structures with minimal effort. The C++ feature that enables the STL are template classes which are also not available in Java. One of the main arguments for Java is its integrated garbage collection. Garbage collectors for C++ are available[33], but using them in conjunction with PAM-Blox II did not prove to provide any advantage since the compilation and execution of a PAM-Blox II program is to short for garbage collection to make a significant difference.

Why not SystemC[27]? SystemC is optimized for the hardware design process by mirroring the philosophy of simulation languages such as VHDL or Verilog. The goal in this project is not general purpose behavioral synthesis but domain-specific compilation for FPGAs. The difference is *design versus programming*, *simulation and hardware verification versus trial-and-error*, and *IP generation versus open software*. In other words, there is no free SystemC compiler for FPGAs that allows precise control over all the details of the resulting circuit, as required by ASC.

PAM-Blox II is part of *A Stream Compiler (ASC)*, currently under development at Bell Labs, which uses gate level PamDC and integrates extensions to PamDC, module generation (PAM-Blox II), and architecture generation all in a single C++ program. This C++ program can be set to simulate the hardware on the algorithm level or on the gate level. In addition, the same executable can generate a netlist for vendor specific CAD tools such as the Xilinx place-and-route tools. Integration of all FPGA compilation layers within the same execution environment enables a high degree of transparency and cross-level optimization.

Our approach is to build up layers of abstraction with a bottom-up approach. The current focus is on integrating module generation and architecture generation. Details on architecture generation will be presented in a future publication.

2. Module Generation in PAM-Blox II

A conventional hardware module library stores the implementations of a large set of hardware modules. *A module generation library* distinguishes itself from a conventional library of hardware modules by storing the algorithm that generates a set of hardware modules based on input parameters such as bitwidth of inputs and outputs, and sign representation of inputs and outputs. An example for the importance of generating different units for the various combinations of bitwidths is the array multiplier which occupies an area of $m \times n$ cells, where m and n are the bitwidths of the multiplicand and multiplier respectively. In this sense, module generation is really a software system which designs hardware, rather than an extension of a hardware description system.

The PAM-Blox II module generation framework contains extensions to the underlying "Gate level to Netlist" layer, PamDC, and an updated methodology for utilizing object-oriented features of C++ to module generation for the purpose of computing with FPGAs.

Towards the upper layers PAM-Blox II contains features to interface the module generation environment to the architecture generation layer of ASC.

Extensions to PamDC:

1. `class Net:` The generic `class Net` encapsulates a set of wires of variable size, and thus enables width inference at the module generator level. This class simplifies the C++ code required to describe the generators. In addition, `class Net` contains a set of user defined operators that further simplify the description of operations on entire sets of wires, such as assignment, indexing and concatenation. A key feature of `class Net` is compatibility with the Standard Template Library of C++, which is not compatible with PamDC objects such as `Bool`s, `Wire`s or `WireVector`s.
2. Support for various *sign representation modes* on the wire level: In order to support multiple sign representations such as twos-complement, sign magnitude, and unsigned numbers, extensions to PamDC handle variable bitwidth assignments of all the supported sign representation modes.
3. Xilinx Virtex support includes wrappers for generating large block RAMs available in the Xilinx Virtex FPGA family as dedicated, parametrizable blocks of memory. The gate level designer has the option to select the width of the constant-size block RAM within the limits of the particular underlying FPGA technology.
4. In order to make the ASC project and PAM-Blox II in particular more accessible, PamDC is ported from Compaq ALPHA cxx to GNU gcc version 2.95.2 or higher. Even though C++ is standardized, porting software between platforms is still a major challenge because most of the C++ compilers do not implement all mandatory features of standard C++. Another cause of difficulties lies within the differences in text (source code) line termination conventions on the various systems.

PAM-Blox II is implemented on top of the extensions to PamDC. **Object oriented features** of C++ correspond to the tasks involved in describing hardware module generators as follows:

1. **Encapsulation** of a module generator in a C++ class: Object state represents the internal wires and parameters of the module. These parameters can be accessed by various other components of the architecture generation environment such as the scheduler or possibly, a high level area and timing estimator. This objects functions (methods) describe the logic parametrically, generating the hardware module based on the input parameters. These parameters can be partially inferred from the input variables (`Net` objects) and their state.
2. **Code-reuse** is implemented by a C++ class hierarchy with explicit inheritance controlled by defining virtual functions and function overloading. Child objects inherit all public methods (functions) and variables (state). For example, all objects with a carry-chain, such as adders, counters, and shifters, inherit the carry-chain definition functions from their common parent. This particular example of code-reuse is paramount to porting the module generators from one FPGA family to another. Details on porting Xilinx XC4000 carry chain generators to Xilinx Virtex devices using inheritance and code-reuse are summarized at the end of this section.

The major changes over the initial PAM-Blox[24] implementation, in addition to the object-oriented design decision mentioned above, are:

1. **Use of template classes:** A template class is a description of a class that can be instantiated with different variable types as inputs. The most common use of template classes is in the Standard Template Library(STL). An STL class such as a `vector` can be instantiated as a vector of integers (`vector<int>`), a vector of floats(`vector<float>`) or a vector of any other user-defined class such as `vector<Net>`. The initial PAM-Blox implementation uses template classes to distinguish hardware integers with different bitwidths as different types, as suggested by PamDC. PAM-Blox II uses `class Net`. As a consequence PAM-Blox II treats variables with different bitwidths as variables of the same type with a different attribute (or object state). The architecture generation layer contains more extensive use of template classes.
2. **An object-specific "enable" for control:** Sequential modules iterating in parallel for a specific number of clock cycles require a control input to coordinate the number of iterations. For example, a one-cycle adder followed by an N cycle sequential multiplication requires separate control lines for the two

units to be pipelined correctly. A priori options were: (1) provide separate clocks, (2) add an enable signal to the logic equations (LUT) of the module, or (3) use the enable input of the flip flop. Providing separate clocks was ruled out early on due to the limitation to very few clock buffers on FPGAs. Enable inputs as part of the object logic (2) are used in the initial PAM-Blox implementation. PAM-Blox II provides a more efficient, separate enable line for flip-flops (3) of each hardware object. Enable lines can now be transparently connected to all flip flops generated within a particular hardware object. A scheduling phase (within the architecture generation layer) connects the enable lines of each object to the control state machine according to the results of the scheduling phase. In C++ these control lines are handled in a hierarchical fashion which greatly simplifies the design of the architecture generation layer. The advantages gained from this feature within PAM-Blox II are a direct result of the tight integration and joint development of the module generation and architecture generation layers.

3. Adapting the design of PAM-Blox II to the limitations posed by gcc: Since gcc is changing rapidly, the details of the limitations of the current gcc implementation are important but not directly relevant to this paper.

In summary, a PAM-Blox II hardware object state consists of: latency, number of sequential cycles, a list of nested sequential objects, a maximal sequential cycle within the object (for nested objects), size (bitwidth), a hierarchical name for debugging, an enable signal, a clock signal, and an "inputs valid" signal.

The module generation environment currently consists of around 170 module generators in about 10K lines of C++ code resulting in an average of less than 60 lines of code per module generator (including white spaces and comments). For comparison, the initial PAM-Blox implementation consists of about 2K lines of code containing 34 module generators.

2.1. Porting XC4000 carry chains to Virtex Devices

Carry chains form the basis of almost all arithmetic circuits from adders, subtracters, multipliers, and dividers, to more specialized units such as counters, comparators, and leading-one-detect circuits.

A conventional binary full adder with inputs A and B has the following well known logic equations:

$$sum_i = A_i \;\; xor \;\; B_i \;\; xor \;\; carry_{i-1} \qquad (1)$$

$$carry_i = (A_iB_i) \; or \; (A_icarry_{i-1}) \; or \; (B_icarry_{i-1}) \qquad (2)$$

For all FPGAs with *a dedicated carry chain*, the above equations have to be mapped to a four input lookup table (the lookup table available for logic in the cell) plus some dedicated custom carry logic. The various FPGA families vary in the precise way that this partition is accomplished.

In order to simplify porting PAM-Blox to new carry chain organizations, the two equations above are described by two separate *virtual functions* that can be overloaded and inherited. The next step lies within the details of the partition of the carry chain equations for the two technologies at hand, Xilinx XC4000 and Xilinx Virtex devices.

The starting point for porting netlists to new FPGA families is the Xilinx library guide which describes the Xilinx library modules. In addition, the data sheet for the particular device provides the complementary pieces of information. From these documents we learn that for Xilinx XC4000 FPGAs the equations for addition in C++ become:

```
    sum[i] = A[i] ^ B[i] ^ carry[i-1];    (3)
```

```
carry[i] = (A[i]&B[i])|(A[i]&carry[i-1])|
           (B[i]&carry[i-1])=             (4)
         = mux(A[i]^B[i],carry[i-1],ZERO);
```

For Xilinx XC4000 devices, the dedicated carry chain is inferred by the Xilinx place and route tools based on relative placement constraints that lock the particular wires to positions relative to each other.

For Virtex devices the equations for addition are:

$$\mathtt{sum}[\mathtt{i}] = \mathtt{xorcy}(\mathtt{LUT}[\mathtt{i}], \mathtt{carry}[\mathtt{i}-1]); \qquad (5)$$

$$\mathtt{carry}[\mathtt{i}] = \mathtt{muxcy}(\mathtt{LUT}[\mathtt{i}], \mathtt{ZERO}, \mathtt{carry}[\mathtt{i}-1]); \qquad (6)$$

Function calls `muxcy(select,input1,input0)` and `xorcy()` instantiate dedicated carry chain logic primitives available inside the Virtex logic blocks. The `LUT[]` array describes the logic that goes into the Virtex adders lookup table. In the case above the lookup table holds the exclusive-or of the two inputs, or `LUT[i]=A[i]^B[i]`. Since carry chains use dedicated blocks explicitly, there is no need for relative placement constraints to infer a carry chain such as is necessary for XC4000 FPGAs.

Since the only difference between the two technologies is in the above two equation, declaring each one of these equations in a separate virtual function enables porting PAM-Blox II by overloading the carry chain functions of the the top ancestor class. *Thus, partitioning the logic into appropriate virtual functions is the key to portability of an object-oriented module generation environment and also provides one the key advantages for using object oriented technology.*

In addition to Xilinx FPGAs, we also consider porting PAM-Blox II to Altera devices. However, any such effort is

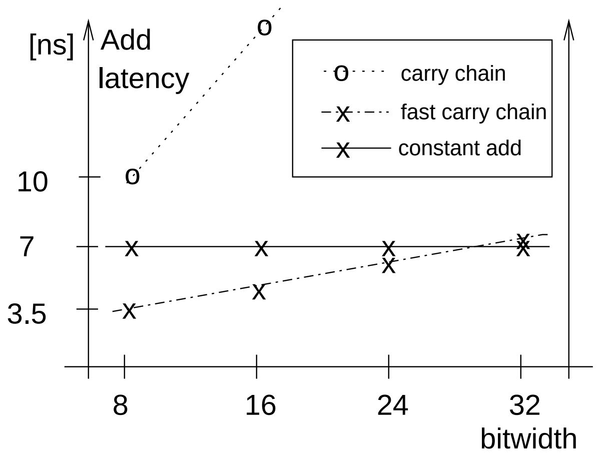

Figure 2: The figure shows latency of addition given three implementation choices: carry chain, (dedicated) fast carry chain, and constant time addition.

complicated by the complete incompatibility of Xilinx and Altera netlists on the `and/or` gate level, even though both netlists are in standard EDIF format. As a consequence this effort is left for future work.

3. Examples of PAM-Blox II Module Generators

In order to show the utility of custom designed module generators for computing with FPGAs, we explain the design of a few sample module generators, and the impact of having such custom modules available in the module generator library. The tradeoffs for the module generators are based on trading area for speed, hand-optimizing technology mapping to the specific FPGA microarchitecture, and utilizing a redundant number representation.

One way of looking at these tradeoffs is by contrasting bit-level flexibility versus a coarse grain programmable architecture. From a different viewpoint, the advantages of optimized module generators present a case for bottom up design of a compiler for FPGAs as opposed to a brute force top down compiler that maps to conventional ALU functionality.

The results for latency and area are based on Xilinx VirtexE devices (speedgrade -6), and standard Xilinx Foundation series v3.2 place and route tools.

3.1. Addition and Subtraction

Addition and Subtraction are the most important module generators for computing with FPGAs. The results in this section quantify the advantages of the FPGAs fast carry chain versus redundant representations.

3.1.1. Using Redundant Representations

Redundant representations are one of the key methods to speed up arithmetic circuits in VLSI[34]. Redundant encodings are defined by Omondi[36] page 456, as: "A radix-R redundant signed-digit number system is one that is based on a digit set:

$$S \triangleq \{-N, -(N-1), \ldots, -1, 0, 1, \ldots, M-1, M\}, \quad (7)$$

where $1 \leq N \leq R-1$, $1 \leq M \leq R-1$ and $|S| > R$. The last condition allows each digit to assume more than R values and gives rise to redundancy."

Such redundant digits enable us to trade off area (more bits) for time by eliminating the carry chain and obtaining "constant time addition", where addition time does not depend on the bitwidth of the operands.

Figure 2 compares carry chain adders with and without the dedicated fast carry chain, and a constant time adder using the redundant digit set $d_i \in \{0, 1, 2\}$ with $R = 2$. Such a digit set requires two bits to represent each digit and, thus, results in a doubling of the required bits to represent a value. The graph in figure 2 shows the order of magnitude speedup of carry chain addition provided by the Xilinx fast carry chain. A single redundant addition is comparable to a 32 bit carry chain add. Despite that fact, a collection of adders such as present in an array multiplier (results in figure 3) shows significant time savings for redundant adders even for bitwidths smaller than 32 bits. Interestingly, not only does the redundant implementation outperform the multiplier with fast carry chains, but even scaling turns out to work in favour of redundant digits resulting a smaller slope of the redundant multiplication line in figure 3. *This surprising result is due to the structure of redundant representations.* Most of the delay is in the interconnect to and from the unit. By placing multiple units together, Xilinx place and Route tools can minimize this interconnect delay and thus optimize the performance of the combined circuit.

As for area, redundant multipliers are about 5% smaller than carry chain based implementations. The area advantage results from a slightly higher utilization of FPGA resources due to technology mapping of conventional $(3, 2)$ counters[35] which are the basic building blocks for computing with a redundant representation. A more detailed description of redundant arithmetic is beyond the scope of this paper.

A further optimization of multipliers for computing with FPGAs can be applied to constant multiplication, as shown in a previous paper[24].

3.1.2. A Sign-Magnitude Add/Sub Unit

PAM-Blox II currently supports three sign modes which are also supported by the architecture generation level of ASC:

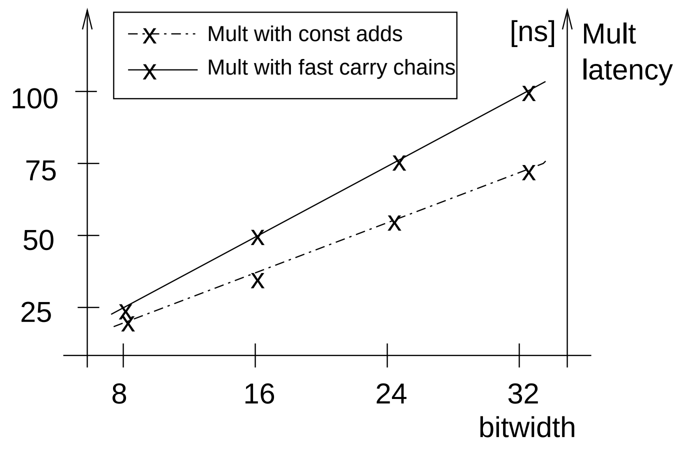

Figure 3: The figure shows a latency comparison of two multiplier implementations: with (dedicated) fast carry chains, and with internal redundant representation.

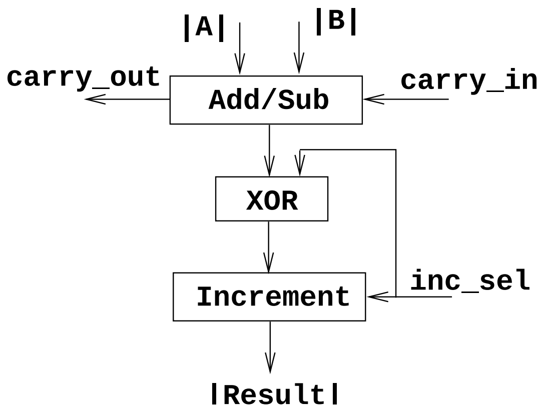

Figure 4: The figure shows the implementation of a Sign-Magnitude adder.

`UNSIGNED`, `TWOSCOMPLEMENT`, and `SIGN MAGNITUDE`. Since the implementation of sign magnitude addition and subtraction proved more challenging than expected, we show some details of the module generator for such sign magnitude addition as an example for PAM-Blox II units.

Figure 4 shows the detailed microarchitecture of a unit that adds or subtracts two numbers `A`,`B` that are represented by sign bits `signa, signb`, and magnitudes |`A`|, |`B`|. The source code in figure 5 shows the implementation of the module generator.

3.2. Comparison `operator==`

A common computational element is to check if two values are equal. Looking at the problem in a top down approach one might consider using a subtracter and checking if the result is zero. Given the flexibility at the bit-level there

```
AddSubSM::out() {


 signa = A.get_last();
 signb = (sub ? ˜B.get_last() : B.get_last());

 carry_in = signa ˆ signb;

 // PAM-Blox II base class
 PamComponent *Adder,*Inc;

 // pipelined add/sub unit
 Adder=new AddSub(A.from_to(0,A.size()-2),
                  B.from_to(0,B.size()-2),
                  carry_in,
                  AddSubOut,
                  &carry_out,
                  &clk);

 sel=reg(carry_in,clk) & ˜carry_out;
 // conditional increment
 Inc = new XorIncrement(AddSubOut,
                         inc_sel,
                         OutMag,
                         &clk);

 Adder->out();    // instantiate logic
 Adder->place();  // relative placement

 Inc->out();   // instantiate logic
 Inc->place(); // relative placement

 // assign magnitude to Result
 W.EqualVector(OutMag, Result,
               OutMag->size());

 // generate the sign bit for the result
 Result.get_last()=reg((reg(signa&signb)|
    (reg(signa &˜signb) & carry_out)|
    (reg(˜signa & signb) & ˜carry_out)));


}
```

Figure 5: PAM-Blox II code for sign-magnitude addition and subtraction. Method `from_to()` extracts a bitfield out of a variable. Method `get_last()` returns the most significant bit which in this case is the sign bit. Note that pipeline stages are set explicitly by declaring flip-flops `reg(input, clock)`.

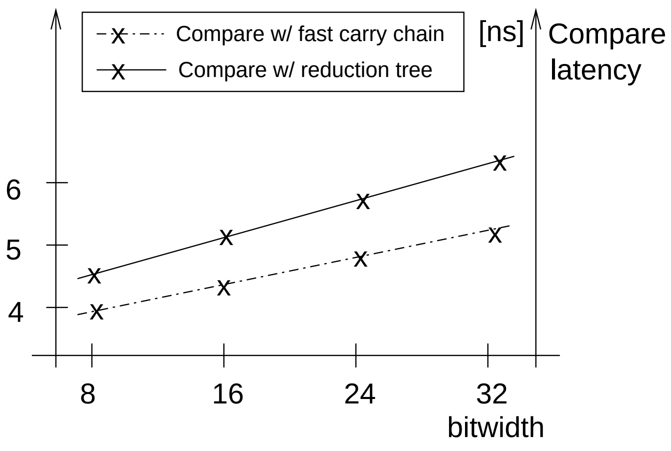

Figure 6: The figure shows two options of comparing a variable with a constant value. (1) with a carry chain, and (2) with a reduction tree.

are two interesting solutions, one for *checking equality of a variable and a constant* and one for *checking equality of two variables*. Optimizing for area and latency respectively, one could implement the comparison operation *(1) with a carry chain, or (2) with a parallel, tree-like implementation.*

A closer look at the implementation for comparing a variable with a constant shows that the carry-chain version can be a subclass of an adder. Such a modified adder then simply requires the overloading of the carry chain's LUT functionality which is a separate virtual function within the adder. The code-fragment below shows one version of the PAM-Blox II code defining a comparison between a variable A and a constant K at bit position i.

```
virtual EquationHandler LUT(int i){
return((((K>>i)&1) ? A[i]  :~A[i])&
   (((K>>(i+1))&1) ? A[i+1]:~A[i+1])&
   (((K>>(i+2))&1) ? A[i+2]:~A[i+2])&
   (((K>>(i+3))&1) ? A[i+3]:~A[i+3]));
}
```

This code implies that a single four-input LUT compares up to four bits against a constant value. As a consequence, the area of the resulting unit is four times smaller than a subtracter and delivers the result of the comparison on the carry out wire of the unit. A similar construction for comparing two variables leads to a unit of half the size of a subtracter.

A *tree-like implementation* still reduces up to four bits per lookup table, but instead of a carry chain, the result is obtained by reducing the input in a tree like fashion. PAM-Blox II code for such a reduction tree is slightly more arduous.

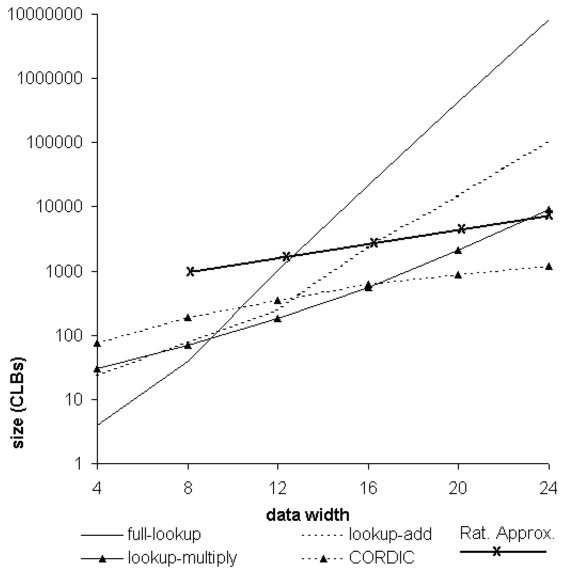

Figure 7: Size comparison in CLBs, for the function evaluation stage with varying data width. Note that values for bipartite tables (lookup-add) are estimated based on computational resources per CLB.

Comparing two variables limits the number of bits that can be compared in one lookup table to two bits of each input variable. As a consequence, for the carry chain solution, comparing two variables takes about twice the area of comparing a variable to a constant, and about half the area of a subtracter.

Figure 6 shows the resulting latencies of comparing variables with constants for a range of input bitwidths. Latency results for comparing variables with variables are similar. From standard VLSI experience we expect a circuit with a hierarchical, or tree based solution to be faster than a carry chain. From an FPGA designers view we expect any solution that uses the fast carry chain to be superior. The results show that the dedicated fast carry chain solution is in fact faster than the hierarchical solution.

One of the conclusions from this result is that knowledge from VLSI design is not directly applicable to FPGA design despite the fact that both are hardware design methodologies.

3.3. Elementary Functions

In a previous paper[37] we present an approach to parameterize pipelined designs for differentiable function evaluation using lookup tables, adders, shifters, multipliers, and dividers. This approach provides an efficient way to develop implementations of function evaluators based on lookup tables, the size and performance of which can be estimated parametrically.

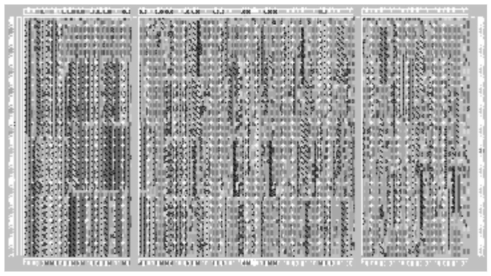

Figure 8: Implementation of IDEA Encryption on an XCV300E FPGA, including glue logic for an Annapolis Microsystems Wildcard.

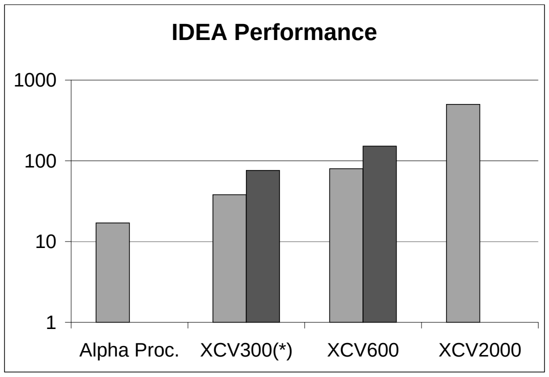

Figure 9: Performance [Mbits/s] of IDEA Encryption on a Compaq Alpha processor and a range of Xilinx Virtex devices. (*)This implementation is run on the Wildcard board.

We compare the following options for a fully pipelined function evaluation unit for the $\log$ function: (1) full lookup tables, (2) a lookup followed by a multiply-add step (lookup-multiply), (3) a lookup followed by addition (lookup-add), (4) shift and add based CORDIC units, and (5) rational approximation. The tradeoff between the various units in terms of area required for achieving full throughput operation, given a target output precision is shown in figure 7.

These results suggest that the optimal microarchitecture for function evaluation depends on the bitwidth of the input and output. For applications with high throughput requirements, these results for area of the various units are the main criterium for deciding on the architecture. We are currently implementing a library in the architecture generation and module generation layers that utilizes these module generators for evaluating elementary functions.

4. Application 1: IDEA Encryption

IDEA encryption serves to demonstrate the effects of the above redundant multipliers on the performance of an application.

The International Data Encryption Algorithm (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. A single algorithm uses different keys for encryption and decryption. The inner loop is repeated eight times, and consists of operations: xor, multiplication, addition, and $(\bmod \quad 2^{16}+1)$.

Figure 9 shows a performance comparison of running the inner loop of IDEA encryption on an Alpha, EV5.6 (21164A) processor operating at 532 MHz, and a series of Xilinx VirtexE FPGAs (speedgrade -6). The FPGA designs include glue logic for the Wildcard [38] from Annapolis Microsystems with a Xilinx XCV300E device. The implementation for the Wildcard (XCV300E) shown in figure 8 utilizing 99% of the CLBs is obtained with a preliminary version of the ASC system.

The performance results show a speedup of about $2\times$ for the conventional XCV300E implementation without “redundant multipliers”, and another factor of two speedup with “redundant multipliers” for the XCV300E and the XCV600E, using the redundant adders from section 3.1. In the case of the XCV2000E the design is fully unrolled and throughput does not depend on the latency of the operations.

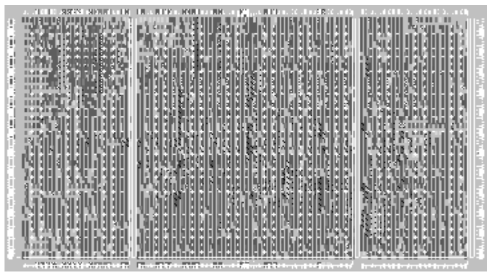

Figure 10: Implementation of LZ compression on an XCV300E FPGA, including glue logic for an Annapolis Microsystems Wildcard.

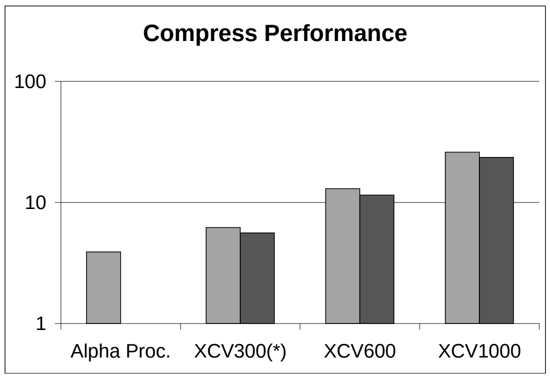

Figure 11: Performance [Mbits/s] of compression on an Alpha processor and a range of Virtex devices. (*)This implementation is run on the Wildcard board.

5. Application 2: LZ Compression

The results for compression demonstrate the effects of optimal comparison units on compression performance.

Lempel-Ziv (LZ) compression has many variations. In this example we implement a very simple form of LZ compression where we look at D bytes of history, and try to match a string up to length D into the future. As a consequence the implementation consists of a two-dimensional array of comparison units.

Figure 11 shows the performance comparison of the variant of LZ compression with $D = 26$, using the same methodology as in the previous example. The implementation for the Wildcard/XCV300E shown in figure 10 utilizes 99% of the CLBs. The results show that the 10% improvement in cycle time of the stand-alone compare units, described in section 3.1, scales to a 10% performance improvement for our variant of LZ compression.

6. Related Work

The commercial module generator library available from Xilinx (CoreGen) contains module generators which can be instantiated through a stand-alone GUI. This approach is very well suited for the CAD tool flow but less ideal for a programming environment. A direct comparison of the performance values from the Xilinx CoreGen datasheets is complicated since the numbers in this paper are real design results, while Xilinx values are maximal (best-case) values.

Pebble[40] a language designed at Imperial College generates VHDL modules for a conventional CAD flow, but requires the user to learn a new language syntax and use the CAD design methodology. The Java Hardware Description Language (JHDL)[31] is a similar effort to PAM-Blox/ASC. Besides the arguments for and against Java, JHDL also integrates module generation with the higher compilation layers. Additionally, JHDL contains a runtime system, a port to Virtex II, and a large set of modules. Similarly, a commercial effort by Celoxica[28] provides the "programming feel" to FPGA design, mostly targetting embedded systems.

One difference of the approach proposed in this paper to these related projects is the emphasis on handling different number representations to tap into the full potential of the FPGAs flexibility on the bit level.

7. Conclusions

A large and flexible module generator library is at the core of computing with FPGAs, enabling the programmer to take full advantage of the bit-level flexibility of FPGAs. Careful utilization of C++ features yields an efficient abstraction for the development, maintenance, and extension of a large module generator library. Concrete conclusions from the sample module generators shown in this paper are:

1. A single redundant (constant latency) addition is comparable to a 32 bit carry chain add. Despite that fact, a collection of adders such as present in an array multiplier shows significant time savings for redundant adders even for bitwidths smaller than 32 bits. This surprising result can be explained by finding that most of the delay of a stand-alone constant-time adder can be optimized away when compiling a whole set of such adders, while the delay through the carry chain is fixed by the technology.
2. Knowledge from VLSI design is not directly applicable to FPGA design despite the fact that both are hardware design methodologies. In particular, design tradeoffs depend largely on the available resources in the FPGA cell, and on the optimality of technology mapping, which can be controlled within the module generation layer.
3. Optimizing the efficiency of elementary function evaluation requires the adaption of the microarchitecture to the precision requirements of the application and the underlying FPGA structure.

We are currently focusing our attention on the architecture generation level. However, we foresee another layer of software on top of architecture generation, which will automate tasks such as precision analysis, loop transformations, memory management generation, and partitioning of an application into software and hardware accelerators. In addition, such a high level transformation layer will be able to deal efficiently with data-structures. The combination of these techniques has the potential to attack the memory wall[39] and result in productive interactions with the microprocessor design community. Clearly, some of the high level transformations will not be fully automate-able in the short run. Some transformations will have to be partially automated in conjunction with user hints. Minimizing the user hints necessary for successful acceleration across a wide range of applications is one of the long term goals.

8. Acknowledgments

We are indebted to Mark Shand for keeping PamDC up to date with current Xilinx technology, for his efforts in porting PamDC to GNU gcc, and for his help with various PamDC related issues. Wayne Luk has been a constant source of encouragement and help. Nicolas Boullis implemented the initial array multipliers, dividers, a generator for the evaluation of the log function.

9. References

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

[2] D. A. Buell, J. M. Arnold, W. J. Kleinfelder, *Splash-2, FPGAs in a Custom Computing Machine* IEEE Computer Society Press, 1996.

[3] 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, Dec. 1997.

[4] O. Mencer, M. Morf, M. Flynn, *Hardware Software Tri-Design of Encryption for Mobile Communication Units* International Conference on Application Specific Signal Processing, Seattle, May 1998.

[5] O. Mencer, M. Morf, *CORDICs for Reconfigurable Computing* The Sixth FPGA / PLD Design Conference and Exhibit, Yokohama, Japan, June 24-26, 1998.

[6] M. Shand, J. Vuillemin, *Fast Implementations of RSA Cryptography*, 11th IEEE Symposium on Computer Arithmetic, Windsor, ONT, Canada, 1993.

[7] F. F. Lee, *A Scalable Computer Architecture for Lattice Gas Simulation*, PhD Thesis, Stanford, June 1993.

[8] H. Styles, W. Luk, *Customising graphics applications: techniques and programming interface*, IEEE Symposium on Field Programmable Custom Computing Machines (FCCM), Napa, CA, April 2000.

[9] S. D. Haynes, P.Y.K. Cheung, W. Luk, J. Stone, *Video Image Processing with the SONIC Architecture*, IEEE Computer, April 2000, pp 50 - 57.

[10] T. J. Callahan, J. R. Hauser, J. Wawrzynek, *The Garp Architecture and C Compiler.*, IEEE Computer, April 2000.

[11] J. Frigo, M. Gokhale, D. Lavenier, *Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective.*, IEEE FPGA Conference, Monterey, CA, Feb. 2001.

[12] M. Weinhardt, W. Luk, *Pipeline Vectorisation for Reconfigurable Systems*, Proc. Int. Symp. FCCM, IEEE Computer Society Press, 1999.

[13] M. Stephenson, J. Babb, S. Amarasinghe, *Bitwidth Analysis with Application to Silicon Compilation*, Proc. of the ACM Conf. on Programming Language Design and Implementation, Vancouver, BC, June 2000.

[14] R. Razdan, PRISC: Programmable Reduced Instruction Set Computers, Ph.D. thesis, Harvard University, Division of Applied Sciences, May 1994.

[15] K. Bondalapati, V.K. Prasanna, *Dynamic Precision Management for Loop Computations on Reconfigurable Architectures*, In IEEE Symposium on FPGAs for Custom Computing Machines, April 1999.

[16] M. Budiu, S. C. Goldstein, K. Walker, M. Sakr, *BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations*, Europar Conf., Munich, Germany, Aug. 2000.

[17] L. Semeria, *Applying Pointer Analysis to the Synthesis of Hardware from C*, Ph.D. thesis, Electrical Engineering Department, Stanford University June 2001.

[18] O.S. Unsal, I. Koren, C. M. Krishna, C. A. Moritz, *Cool-Cache for Hot Multimedia*, MICRO-34 Conference, Austin, Texas, Dec, 2001.

[19] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, S. A. McKee, *The Impulse Memory Controller*, IEEE Trans. on Computers, Nov. 2001.

[20] O. Mencer, H. Huebert, M. Morf and M.J. Flynn, *StReAm: Object-Oriented Programming of Stream Architectures using PAM-Blox*, Field-Programmable Logic and Applications, LNCS 1896, Springer, pp. 595–604, 2000.

[21] M. Gokhale, J. Kaba, A. Marks, J. Kim, *Malleable architecture generator for FPGA computing*, Reconfigurable Logic, Proc. SPIE 2914, Bellingham, WA, Oct. 1996.

[22] S.G. Abraham, B.R. Rau, *Efficient design space exploration in PICO* Proc. CASES 2000 Internatinal Conference on Compilers. Architecture and Synthesis for Embedded Systems, San Jose, California, Nov. 2000.

[23] *The Xtensa Processor* http://www.tensilica.com/

[24] O. Mencer, M. Morf and M.J. Flynn, *PAM-Blox: High Performance FPGA Design for Adaptive Computing*, Proc. IEEE Symp. on FPGAs for Custom Computing Machines, IEEE Computer Society Press, pp. 167–174, 1998.

[25] Synopsys, *http://www.synopsys.com/products/fpga/fpga_express.html*

[26] P. Bertin, H. Touati, *PAM Programming Environments: Practice and Experience*, IEEE Workshop on FPGAs for Custom Computing Machines, April 1994.

[27] J. Kunkel, K. Kranen, *SystemC demonstrates rapid progress*, EE Times, Sept. 2000.

[28] J. Kunkel, K. Kranen, *Celoxica adds simulator, debugger to Handel-C compiler*, EE Times, Feb. 2001.

[29] S. A. Guccione, D. Levi, *XBI: A Java-based Interface to FPGA Hardware", in Configurable Computing Technology*, Proc. SPIE Photonics East, John Schewel, ed., Bellingham WA, Nov. 1998.

[30] A. Frey, G. Berry, P. Bertin, F. Bourdoncle, J. Vuillemin, *Jazz is a high-level programming language for expressing* [. . .] *large digital synchronous circuits.* http://www.exalead.com/jazz/

[31] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, M. Rytting, *A CAD Suite for High-Performance FPGA Design*, IEEE Symposium on Field Programmable Custom Computing Machines (FCCM), Napa, CA, April 1998.

[32] B. Stroustrup, *The C++ Programming Language, 3rd ed.* Addison-Wesley, 1997.

[33] H. Boehm, *Space Efficient Conservative Garbage Collection*, Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices 28, 6, June 1993.

[34] D.S. Phatak, T. Goff, I. Koren, *Constant-Time Addition and Simultaneous Format Conversion based on Redundant Binary Representation*, IEEE Trans. on Comp., Nov. 2001.

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

[36] A. Omondi, *Computer Arithmetic Systems*, Prentice Hall, 1994.

[37] O. Mencer, W. Luk, *Parameterized High Throughput Function Evaluation for FPGAs*, Journal on VLSI and Signal Processing (special issue on field programmable logic), Kluwer Academic Publishers, Netherlands, 2002.

[38] Annapolis Microsystems *Wildcard, a Cardbus based FPGA Accelerator card.* http://www.annapmicro.com/

[39] W. A. Wulf, S. A. McKee, *Hitting the Memory wall: Implications of the Obvious.* Computer Architecture News, 23(1), March 1995.

[40] W. Luk, S. McKeever, *Pebble: a language for parametrised and reconfigurable hardware design.* Field-Programmable Logic and Applications (FPL), Springer, 1998.