PAM-Blox: High Performance FPGA Design for Adaptive Computing

Oskar Mencer, Martin Morf, Michael J. Flynn

Computer Systems Laboratory, Department of Electrical Engineering
Stanford, CA 94305, USA
email: oskar@umunhum.stanford.edu
http://umunhum.stanford.edu/PAM-Blox/

Abstract

PAM-Blox are object-oriented circuit generators on top of the PCI Pamette design environment, PamDC. High- performance FPGA design for adaptive computing is simplified by using a hierarchy of optimized hardware objects described in C++.

PAM-Blox consist of two major layers of abstraction. First, PamBlox are parameterizable simple elements such as counters and adders. Automatic placement of carry chains and flexible shapes are supported. PaModules are more complex elements possibly instantiating PamBlox. PaModules have fixed shapes and are usually optimized for a specific data-width. Examples for PaModules are multipliers, Coordinate Rotations (CORDICs), and special arithmetic units for encryption.

The key difference of our approach to most other design tools for FPGAs is that the designer has total control over placement at each level of the design hierarchy, which is the key to high-performance FPGA design. Second, the object interface was chosen carefully to encourage code-reuse and simplify code-sharing between designers.

PAM-Blox are intended to be part of an open library that allows design sharing between members of the adaptive computing community.

1. Introduction

Adaptive computing has been an active area of research over the past decade. A summary of the current state of the field is given in [3]. While it has been shown that FPGAs can achieve an improvement in performance and power over existing general purpose processors, competitive FPGA designs have been created mostly on a very low, structural level.

The first custom computing machines, the PAM (see section 2) and the Splash-2 [2], were build shortly after the introduction of FPGAs by Xilinx in 1985. Both projects investigated the feasibility of FPGAs as computing platforms.

Conventional general purpose processors consist of a fixed, general data-path, and programmable control (instructions) for that data-path. A few arithmetic units are highly optimized for low latency[7, 8].

On custom computing machines data-path and control are fully programmable, allowing the designer to tailor the architecture of the computer to the structure of the algorithm. Flexibility or reconfigurability comes at the expense of latency (i.e. longer cycle time) and logic density on the chip.

Given today's technology, custom computing machines can compete with general purpose processors on latency tolerant applications that require a relatively small amount of logic. Due to large reconfiguration times of today's devices single FPGAs do not scale well to large problem sizes or large data-flow graphs. Multiple FPGAs can be used to compute larger problems. The major drawback is the very high complexity of partitioning a design onto multiple FPGAs given a limited amount of pins. Overcoming the pin-limitation in software – with design tools – is investigated in the Virtual Wires[17] project. Eliminating the pin limitation with multi-chip modules of FPGAs is explored in the Teramac project[11].

Applications that execute favorably on FPGAs are therefore data intensive applications which can be executed in very deep pipelines (e.g. encryption, pattern matching ,etc.) and applications with a huge amount of fine grain parallelism such as Jacobi relaxation (see section 5.1), and lattice gas simulation[6].

We believe that the reason why custom computing machines have not been commercially successful are the still clumsy programming tools for custom computers. Currently, FPGAs are programmed with CAD tools that have been optimized for hardware design.

Some attempts have been made towards creating programming languages for FPGAs [14, 13]. The drawback of simplifying FPGA design is that most of the performance and significant area are lost in the compilation process. Designs created with today's high level languages can not compete with current microprocessor and compiler technology.

Hand designs are typically more efficient than compiled behavioral descriptions. In order to exploit the efficiency of

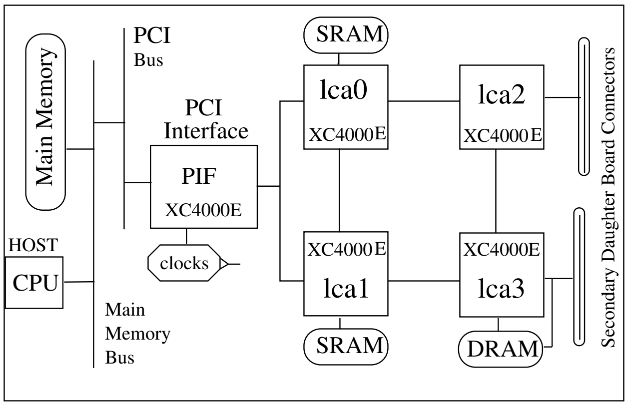

Figure 1: The PCI Pamette board consists of five Xilinx XC4000E FPGAs. One implements the PCI interface. Two are connected to SRAM, 64 KBytes each, and one FPGA is connected to DRAM SIMM sockets. The SRAM memory can be mapped into a continuous piece of main memory on the host system. PCI Pamette can operate in master and slave mode on the PCI bus. A flexible user clock from 360 KHz to 120 MHz can be generated in addition to the recovered PCI clock.

hand design while simplifying the design process, we propose a bottom up approach to compilation for custom computing machines. By creating a powerful and highly optimized parameterizable circuit generators, PAM-Blox serve as an additional level of abstraction that preserves optimal area and performance. Section 2 contains some more details on the history of the PAM project. Section 3 gives an overview of the current state of CAD tools for FPGAs. Section 4 introduces PAM-Blox consisting of PamBlox and PaModules. Section 5 shows a comparison of performance and area of PAM-Blox designs to behavioral synthesis tools. Finally, we conclude with the current status and future work in section 6.

2. PAM-Story

PAM stands for Programmable Active Memories. The first PAM, PeRLe-0, developed at DEC PRL in France[1], is one of the first custom computing machines. Next to the hardware efforts the PAM team also developed a C++ class library, PamDC, for creating designs for Xilinx FPGAs. The most impressive result obtained with a PeRLe-1 board is RSA encryption at Cray speeds[10].

2.1. PCI Pamette

The most current PAM is the PCI Pamette board developed by Mark Shand[15]. The PCI Pamette consists of 5 Xilinx XC4000 series FPGAs.

The highlights of the board and runtime environment include: PCI write-burst capability, a flexible DMA engine, compatibility with DIGITAL Alpha running Unix and Intel x86 PCs with WindowsNT. On both platforms, control of the board is achieved via system calls implemented in the respective device driver.

The PCI Interface (PIF) developed for the PCI Pamette maps a region of main memory to the Pamette board. Communication bandwidth between the host CPU and the FPGAs is set by the clock speed and bus-width of the PCI bus. PCI Pamette supports 32 and 64 bit PCI at 33 and 66 MHz.

The four user programmable FPGAs can be reconfigured individually or in parallel at runtime, allowing the designer to explore dynamic reconfiguration within the limits of Xilinx XC4000 FPGAs[1].

The FPGA design can be clocked by the PCI clock and by an additional user programmable clock. For debugging purposes, the user clock can be single stepped.

The interconnect on the PCI Pamette board is a 2D mesh shown in Figure 1. The signal delay from one FPGA to another is therefore very small, compared to a more flexible switch based interconnect.

3. Existing CAD for Xilinx FPGAs

In this section we describe a few popular CAD tools for Xilinx FPGAs. We present examples for the different approaches to create netlists for FPGAs and relate them to the needs of custom computing environments.

Synopsys FPGA Compiler is part of the CAD environment that is mostly used for ASIC development. FPGA Compiler maps a design to the Xilinx Netlist Format (XNF). For high-level behavioral synthesis the user goes through the Behavioral Compiler, the Design Compiler, the FPGA Compiler, and finally through Xilinx place-and-route tools. With this tool-flow, many independent tools transform the design. At the end it is almost impossible to predict which effect a small change in the initial code will have on the actual design. This problem is made worse by the fact that Synopsys tools do a careful technology mapping, but before Xilinx tools get started, the design is flattened, eliminating much of the work done by the Synopsys tools.

Synopsys FPGA Express is a unified tool compiling Verilog or VHDL to XNF. While the entire process is optimized for FPGAs, the tool still does not allow the designer to gain much insight into the results of the compilation. Like FPGA Compiler, FPGA Express has a powerful state-minimization algorithm for state-machines. In general, data-path performance is suffering from non-optimal automatic placement and high cycle times compared to hand-designed circuits.

[1]For the PCI Pamette board reconfiguration takes about 100 ms.

The second major approach is to use graphical design entry such as ViewLogic. While graphical design entry works well for small designs, large projects are more manageable with hardware description languages such as VHDL or Verilog.

The Trianus system [9] developed at ETH in Zurich, offers an integrated FPGA design system with placement hints at the design entry level. The tool handles everything from technology mapping to place and route. Trianus is currently only available for Xilinx XC6000 series FPGAs. Experience with Trianus shows a significant reduction in compile time, when placement hints are available.

The programmer of a custom computing machine requires a software design-like environment with a short compilation cycle. In addition, there is a need for the equivalent of system calls and library functions.

3.1. PAM Design Compiler: PamDC

PamDC was developed as part of the PAM project[1] described above. The design is described structurally in C++. Running the resulting program creates a Xilinx netlist file which is passed on to place-and-route tools.

The advantage of PamDC is that the designer has full control over placement. Technology mapping can be done automatically or by the designer. This is especially important in order to make efficient use of the fast carry chain, available in Xilinx XC4000 FPGAs. PamDC gives the designer total control over the design, which with some effort can result in maximal performance and minimal area.

The drawbacks of PamDC are the relatively high effort needed to create structural designs on a very low level. In software terminology, PamDC could be compared to assembly level programming.

4. *PAM-Blox*: PamBlox and PaModules

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.

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. We use *PAM-Blox* as the name for the entire design environment. PamBlox stands for templates of hardware objects while PaModules are objects with a fixed size.

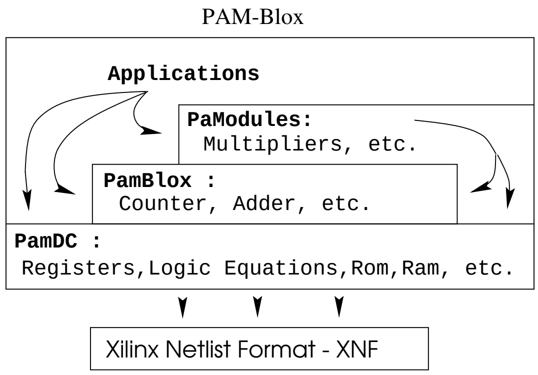

Figure 2: The figure shows the layers of the PAM-Blox design environment. PamDC compiles the design to the Xilinx Netlist Format XNF. PamBlox are interacting with PamDC objects, PaModules interact with PamBlox and PamDC, and the application can access features from all three layers below.

PAM-Blox simplify the design of data-paths for FPGAs by implementing an object-oriented hierarchy in PamDC / C++. With PAM-Blox, hardware designers can benefit from all the advantages of object-oriented system design that the software industry has learned to cherish during the last decade. Efficient use of function overloading, virtual functions and templates make PAM-Blox a very powerful and yet simple to use design environment.

By implementing PAM-Blox together with the actual design within a C++ hierarchy, we simplify the task of adapting library modules to the specific needs of the design. Therefore the PAM-Blox generator library is easily scalable and allows FPGA designers to share and reuse pieces of designs by writing new PamBlox and PaModules.

PAM-Blox are compatible with all the platforms that support PamDC. Currently these are WindowsNT with Microsoft Visual C++ and DIGITAL Alpha workstations with DIGITAL Unix and DIGITAL C++.

4.1. Hardware Objects

There are many ways to describe hardware objects in an object-oriented sequential language like C++. The description of the hardware object has to be efficient in terms of code-size, in order to minimize the complexity of the description. The second major design issue is to create a very explicit specification of the interface to the hardware object i.e. object size, inputs, outputs and optional inputs or outputs.

Figure 3 shows the structure of a hardware object described in C++. All data and methods inside a hardware object are declared public in order to allow maximal visibility during simulation. Inputs and outputs are syntactically sep-

```
class HWobject:public parent{
public:
  <internal wire declarations>

// constructor
  HWobject(input parameters, optionals){
    <initialization of inputs>
  }

  out(output parameters, optionals){
    <internal logic>
  }

  <additional methods called by 'out'>

  place(absolut placement parameters){
    <absolut placement>
  }
  place(){
    <relative placement>
  }
}
```

Figure 3: A general PAM-Blox hardware object described in C++.

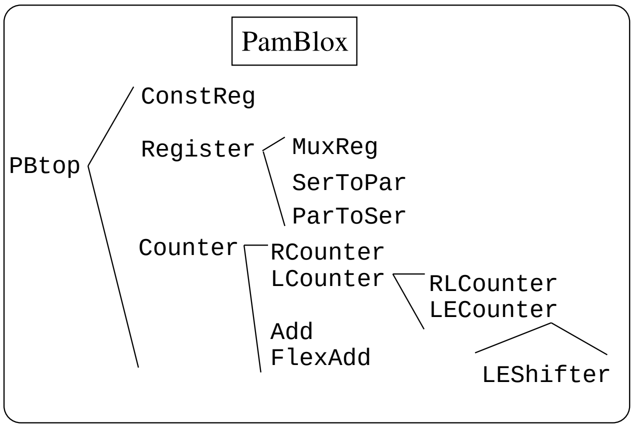

Figure 4: A subset of the PamBlox hierarchy. The top object PBtop consists of minimal logic and a set of placement functions which are inherited by all PamBlox objects. Prefix 'R' stands for "Resetable", 'L' for "Loadable", and 'E' for an "Enable".

arated by passing inputs to the constructor and outputs to the 'out' method. Required parameters are passed by declaring the formal arguments to be type reference, while optional parameters are passed as pointers.

Examples of optional inputs are the clock and the object name. Without specifying a clock, PAM-Blox uses the default clock. An example of an optional output is the carry-out of an adder.

Size is specified as a template parameter. As a consequence the PamBlox interface is protected by C++ type checking.

4.2. PamBlox and PaModules

PamBlox are parameterizable simple templates of objects such as counters, shifters or adders, possibly containing a carry chain. The initial PamBlox hierarchy is shown in Figure 4.

The top object, PBtop, consists of a vector of registers, and a set of placement functions which handle different carry-chain configurations. As an example of code reuse, every child of PBtop inherits the placement functions and can overwrite them if necessary.

PaModules are complex, fixed circuits implemented as C++ objects. PaModules consist of multiple PamBlox and are optimized for a specific data-width. Examples are constant(k) coefficient multipliers (KCMs), Booth multipliers, Coordinate Rotation Digital Computer (CORDIC) circuits [5], and special purpose arithmetic units such as a constant multiply modulo $(2^{16}+1)$ operation for encryption[4].

Figure 5 shows the instantiation of PamBlox within a PaModule object. The PaModule implements a 16 bit multiplier by using a 16x20 bit lookup-table and a shift-accumulate unit. The circuit multiplies 4 bits at a time and accumulates the result.

The table in figure 6 shows the current code-size of the PAM-Blox v1.0 circuit generators. Code-size is given in PamDC / C++ lines necessary to implement the objects.

4.3. Hierarchical Naming

PamDC enables direct control over the naming of wires. PAM-Blox are implemented to support a hierarchical naming scheme that creates a unique name for each wire in the design similar to paths in a filesystem. The name of each wire contains all the ancestors (parent objects) of the wire. The top name can be specified by the designer, e.g. a PaModule multiplier with the name "multy" containing one adder with a carry-chain, would result in the following name for the third element of the carry-chain:

$$multy/add0/carry < 3 > \quad (1)$$

The naming scheme enables designers to use additional tools for debugging and still be able to trace the source of each wire. For example, the naming hierarchy is preserved for simulation (within PamDC) and low-level tools such as Xilinx fplan.

```
// PaModule 16 bit KCM [28 CLBs]
class NibbleMult28:public PMtop{
 public:
<initializations>

void out(
  WireVector<Bool, INPUT_WIDTH>&Out,
  WireVector<Bool, MULT_WIDTH> *SOut=NULL)
{
  int n,i,rval;
  char nmult[4];
  unsigned short romval[ADD_WIDTH];

  // values for 20 LUTs
  for(n=0;n<ADD_WIDTH;n++){
  // 16 bits per LUT
    romval[n]=0;
    for (i=0;i<MULT_WIDTH;i++){
      if ((((factor*i)>>n)&1)==1)
        romval[n]=romval[n] |
               (((unsigned short)1)<<i);
    }
  }
  //create LUT ROMs
  for(i=0;i<ADD_WIDTH;i++)
    PSum[i]=reg(rom(NIn,romval[i]),clk);

  //2nd stage: PamBlox Accumulator
  Add<ADD_WIDTH> *A2;
  A2=new Add<ADD_WIDTH>
          (PSum,RSum,zero,&clk);
  A2->out(result,cout3);
  A2->place();

  // feedback
  for(i=0;i<ADD_WIDTH;i++){
    if (i<16){
       RSum[i]=mux(Nstart,
                 ZERO,
                 result[INPUT_WIDTH+i]);
    }else if (i==17){
      RSum[i]=mux(Nstart,ZERO,cout3);
    }else{
      RSum[i]=ZERO;
    }
  }
}
<placement>
};
```

Figure 5: The code above shows the PaModule implementation of a high performance constant(k) coefficient multiplier (KCM) in 28 CLBs with PamBlox (see section 5.2.).

	PamBlox	PaModules
No. of Objects	28	6
Lines of Code	1370	750
Av. Lines per Object	~ 50	~ 120

Figure 6: The table shows the current size of the PAM-Blox v1.0 circuit generators, given in lines of PamDC / C++ code.

5. Performance and Area

Our metrics of comparison are *minimal cycle time* , *area requirement* in configurable logic blocks (CLBs) and *compile time* (from C++ / PAM-Blox to the Xilinx Netlist Format).

A simple method to estimate the power consumption of an FPGA design is shown in [19]. We use:

$$InternalPower \propto numberofCLBs * Frequency \quad (2)$$

$$=> (Performance/Power) \propto CLBs^{-1} \quad (3)$$

Compile time was measured on a conventional Intel Pentium PC running at 120 MHz. PAM-Blox are compiled with Microsoft Visual C++ 4.0.

Another point of consideration is programmability. It is much easier to create FPGA designs on the behavioral level; however experience shows that for most non-trivial applications much of the performance advantage of FPGAs over microprocessors is lost. Once timing and power constraints can be met by a general purpose DSP processor, programming becomes easier by orders of magnitude.

The objective of this section is to show how much can be gained by focusing on the structural design of data-paths for FPGAs using hand-optimized library elements.

We compare the original implementations of the RAW benchmarks [12] Jacobi, Matrix Multiply and DES encryption synthesized by Synopsys FPGA Express II with implementations using PAM-Blox. The PAM-Blox implementations of these RAW benchmarks have been designed by trying to keep the design effort within order of magnitude to the design effort for the implementations in Verilog.

5.1. RAW: Jacobi Relaxation

Jacobi relaxation is an iterative method for solving differential equations of the form:

$$\nabla^2 A + B = 0 \quad (4)$$

The basic operations for this benchmark are shift and add. The implementations compared in table 7 consist of a 4x4 array with 2x2 active cells and 8 bit values. During

	Compile Time	Area [CLB]	Frequency
JACOBI 4x4 (8 bit)			
FE II	80 s	164	30 MHz
PAM-Blox	45 s	129	35 MHz
DES (1)			
FE II	1,510 s	828	15 MHz
PAM-Blox	86 s	398	22 MHz
MATMULT 4x4 (8 bit)	Compile Time	Area [CLB]	Mega-mmps
FE II	350 s	609	0.38
PAM-Blox 1	77 s	604	1.23
PAM-Blox 2	98 s	954	1.52

Figure 7: RAW benchmarks compiled with Synopsys FPGA Express II (FE II) are compared to PAM-Blox implementations. Compile time stands for the time to compile a design description to a Xilinx netlist file. FPGA Express results are reported for the completely placed and routed system. The performance for matrix multiply is given in matrix multiplications per second (mmps).

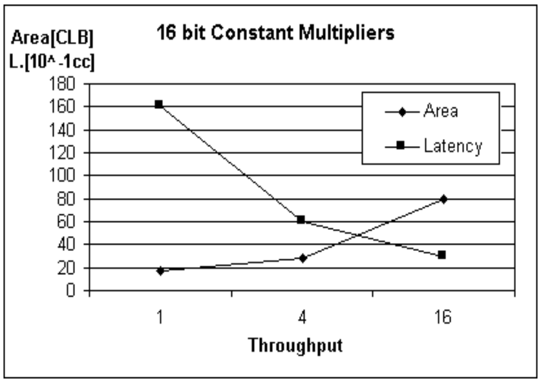

Figure 8: The figure shows Area and Latency for 16 bit constant(k) coefficient multipliers with a throughput of 1,4 and 16 bits per clock cycle. Latency is shown in units of 10^{-1} clock cycles i.e. 160 means 16 clock cycles. Throughput is given in 'clock cycles between successive results'.

each clock cycle, each active cell takes the values of cells neighboring east, south, west and north, adds them together and divides the result by four.

Because the arithmetic operations shift and add map easily onto the Xilinx XC4000E library used by FPGA Express II, there is not much room for area improvement. Clock frequency of the PAM-Blox design is only about 15% higher than the design optimized by Synopsys HDL compiler. The improvement in area is about 20%.

5.2. RAW: DES Encryption

DES encryption is very well suited for implementation in hardware. The basic primitives are fixed permutations and exclusive-or. The results for the PAM-Blox DES design in table 7 show a 50 % improvement in clock frequency with half the area over the original RAW implementation with Synopsys FPGA Express II.

The superior results obtained with PAM-Blox are due to partially manual placement and technology mapping, i.e. the careful design of logic that fits into 4 bit lookup tables and efficient usage of registers.

Further improvement of the throughput of DES could be achieved by pipelining the design. Pipelining on the CLB level maximizes register utilization and would result in maximal throughput.

5.3. RAW: Integer Matrix Multiply

The Matrix Multiply benchmark multiplies two 4x4 matrices with $4^2 = 16$ multipliers and an adder tree.

FPGA Express II uses simple bit-serial shift-and-add multiplication. A full matrix multiply therefore takes more than 50 clock cycles. For this benchmark we chose to create a more efficient computational structure to show how PAM-Blox can be used to adapt the arithmetic units to the specific requirements of the application.

By implementing multiple bit-serial multipliers using Booth encoding, we are able to trade area for performance. Obviously Booth multipliers are more efficient for this specific application. The point is rather to show how the PAM-Blox environment can be used to choose the right arithmetic unit for a specific application.

Table 7 shows two PAM-Blox designs differing only in the selection of the multiplication algorithm[2]. PAM-Blox 1 multiplies the matrices in 27 clock cycles while PAM-Blox 2 takes 19 clock cycles for a full 4x4 matrix multiplication including data transfer. Clock cycle times for the PAM-Blox designs are around 33 MHz. The original design synthesized with FPGA Express II runs at 15 MHz and requires 39 clock cycles for a full matrix multiply. The table above shows the throughput in matrix multiplications per second (mmps). With the right multiplier we see an increase in throughput of up to 4 times, compared to the original RAW benchmark compiled with Synopsys FPGA Express II.

5.4. Constant(K) Coefficient Multipliers - KCM

Constant(K) Coefficient Multipliers (KCMs) are of interest for many applications including filters and encryption. KCMs are implemented as PaModules. We compare 16 bit

[2]for details on the specific multiplier architectures see the PAM-Blox distribution at http://umunhum.stanford.edu/PAM-Blox/ or email pam-blox@umunhum.stanford.edu

KCMs with a throughput of 16, 4 and 1 bits respectively, in figure 8, in order to show the time-space tradeoff for KCMs on Xilinx XC4000 FPGAs. With increasing throughput, we increase the area requirement and decrease latency – trading area for latency and throughput.

First, we implemented the fully-parallel KCM proposed in[18] to PAM-Blox, achieving the same performance and area values reported in the application note from Xilinx. Second, we created an additional design for a 16 bit KCM which takes 4 bits at a time at about $1/3$ the area of the fully parallel version. This multiplier is based on distributed arithmetic [16], combining multiply-adds into lookup tables.

While performance over area for this multiplier is worse than for the fully parallel case, the small area of this multiplier allows us to map the entire IDEA encryption algorithm onto around 3200 CLBs or 4 XC4020E chips resulting in 528 Mbits/s of maximal encryption speed. This design on 4 Xilinx XC4020E-3 was compared to current microprocessors in [4]. Performance over power or MOPS per Watt of the PAM-Blox design on low-power XC4000XV FPGAs is about 6 times higher than the microprocessor based solutions.

The relatively small size of the bit-serial KCM (17 CLBs) allows us to fit more than 45 such multipliers on a Xilinx XC4020E with 800 CLBs.

5.5. Performance of Xilinx Place-and-Route

We expected manual placement to decrease place-and-route time. Instead we found that Xilinx place-and-route performance is dominated by routing. Place-and-route performance therefore varied depending on how easy or hard it is to route the placed design.

While hand-placement managed to improve circuit performance, place-and-route times varied depending on the specific design, FPGA size and seed number for the non-deterministic place-and-route algorithm.

6. Conclusions, Current Status and Future Work

One of the goals of adaptive computing is to offer higher performance with lower power consumption than general purpose processors. We believe that today's FPGA compiler technology enables us to compete with today's general purpose processors only when using a hand-optimized library of FPGA circuits.

We have shown that it is possible to outperform today's FPGA-CAD environments by using PAM-Blox for hand-design of circuits and placement. We believe that one of the reasons for better performance with PAM-Blox is complete control over placement at every level of the design hierarchy – a fact generally emphasized by experienced FPGA designers. We do not claim that PAM-Blox designs automatically result in better performance than high level synthesis. Rather, for data-path intensive applications, an experienced designer can get access to all the features of Xilinx XC4000 FPGA.

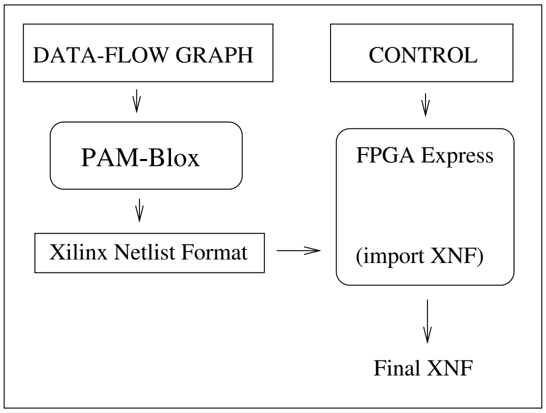

Figure 9: The figure shows one possible tool-flow of a unified design environment enabling control circuits designed with Synopsys FPGA Express to be combined with data-paths designed with PAM-Blox.

Once the PamBlox or PaModules object is created, it is very simple to share it's design with other researchers or reuse portions of their design for other objects.

The object-oriented features of PAM-Blox lead to reduced complexity of the description of a circuit, and fast compilation times lead to shortened design cycles – especially in a team-oriented environment.

PAM-Blox are currently expanded to include a wide variety of arithmetic unit generators from distributed arithmetic to online arithmetic and CORDICs[5]. The objective is to introduce an open repository of firmware to a field which is dominated by proprietary designs.

We would like to encourage sharing of designs by enabling third parties to use and contribute to PAM-Blox. PAM-Blox are therefore distributed under a GNU license, allowing and encouraging anybody to use, modify and redistribute PAM-Blox as long as the GNU license is preserved.

In order to allow the designer to integrate optimized state machines created by traditional CAD tools with a PAM-Blox data-path, PAM-Blox supports easy module generation. These modules can be imported for example into FPGA Express II. A sample tool-flow is shown in figure 9. Consequently the FPGA designer can optimize the data-path with PAM-Blox, while synthesizing the controlling state-machine from a higher level – a methodology widely accepted for high-performance Application Specific Integrated Circuits.

7. Acknowledgments

This research is supported by DARPA Grant No. DABT63-96-C-0106. We would like to thank Mark Shand and Stephan Ludwig of DIGITAL Systems Research Center in Palo Alto for their help and support. We would also like to thank Andre DeHon for sharing some preliminary research results from the BRASS project at Berkeley. DES encryption was implemented by Hon-Cheong Leung. Matrix multiplication and multipliers were implemented by Hyuk-Jun Lee.

8. References

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

[2] Duncan A. Buell, Jeffrey M. Arnold, Walter 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 Magazine, December 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, Pacifico Yokohama, Yokohama, Japan, June 24-26, 1998.

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

[7] H. Al-Twaijry with M. Flynn *Area and Performance Optimized CMOS multipliers* PhD Thesis, Stanford, Aug. 1997

[8] S. Oberman with M. Flynn *Design Issues in High Performance Floating Point Arithmetic Units* PhD Thesis, Stanford, Jan. 1997

[9] Stephan Gehring, Stefan Ludwig, *The Trianus System and it's Application to Custom Computing*, 6th International Workshop on Field-Programmable Logic and Applications, FPL, September 1996.

[10] Mark Shand, Jean Vuillemin, *Fast Implementations of RSA Cryptography*, 11th IEEE Symposium on Computer Arithmetic, Windsor, ONT, Canada, 1993

[11] W.B. Culbertson, R. Amerson, R.J. Carter, P. Kuekes, G. Snider, *Defect Tolerance on the Teramac Custom Computer*, IEEE Symposium Field-Programmable Custom Computing Machines, Napa Valley, CA, April 1997.

[12] Jonathan Babb, Mathew Frank, Victor Lee, Elliot Waingold, Rajeev Barua, Michael Taylor, Jang Kim, Sirkrishna Devabhaktuni, Anant Agarwal, *The RAW Benchmark Suite: Computation Structures for General Purpose Computing*, IEEE Symposium Field-Programmable Custom Computing Machines, Napa Valley, CA, April 1997.

[13] Maya Gokhale, Ron Minnich, *FPGA Computing in Data C*, Proc. IEEE Workshop on FPGAs for Custom Computing Machines, pages 94-101, Napa, CA, 1993.

[14] S. Guccione, M. Gonzalez, *A data-parallel programming model for reconfigurable architectures*, Proc. IEEE Workshop on FPGAs for Custom Computing Machines, pages 79-87, Napa, CA, 1993.

[15] Mark Shand, *The PCI Pamette FPGA board at DEC Systems Research Center*, http://www.research.digital.com/SRC/pamette/

[16] Stanley White, *Applications of Distributed Arithmetic to Digital Signal Processing*, IEEE ASSP Magazine, p4-21, July 1989

[17] Jonathan Babb, Russell Tessier, Anant Agarwal, *Virtual Wires: Overcoming pin limitations in FPGA-based logic emulators.* Proc. IEEE Workshop on FPGAs for Custom Computing Machines, pages 142-151, Napa, CA, 1993.

[18] Ken Chapman, *XApp054: Constant(K) Coefficient Multipliers for the XC4000E*, http://www.xilinx.com/xapp/xapp054.pdf

[19] Xilinx, *Application Brief: A Simple Method of Estimating Power in XC4000XL/EX/E FPGAs*, http://www.xilinx.com/xbrf/xbrf014.pdf