High Level Compilation for Fine Grained FPGAs

Maya Gokhale
David Sarnoff Research Center
maya@sarnoff.com

Edson Gomersall
National Semiconductor
edson@berlioz.nsc.com

Keywords: FPGA, configurable computing, silicon compiler, FPGA design tools, high-level synthesis

1 INTRODUCTION

Over the past several years, Field Programmable Gate Arrays (FPGAs) have functioned effectively as specialized processors capable of an order of magnitude improved performance over workstations at a fraction of the cost. It is widely recognized, however, that for FPGAs to gain acceptance in the software community as algorithm accelerators, tools to create hardware realizations of those algorithms must be greatly improved. In this paper, we present an integrated tool set to generate highly optimized hardware computation blocks from a C language subset. By starting with a C language description of the algorithm, we address the problem of making FPGA processors accessible to programmers as opposed to hardware designers.

Our work is specifically targeted to fine grained FPGAs such as the National Semiconductor CLAy^{TM} FPGA family. Such FPGAs exhibit extremely high performance on regular datapath circuits, which are more prevalent in computationally oriented hardware applications[1] dense packing of datapath functional elements makes it possible to fit the computation on one or a small number of chips, and the use of local routing resources makes it possible to clock the chip at a high rate. By developing a lower level tool suite that exploits the regular, geometric nature of fine grained FPGAs, and mapping the compiler output to this tool suite, we greatly improve performance over traditional high level synthesis to fine grained FPGAs. Individual arbitrary bit-length functions in our library show a factor of 2.6-7.9 improvement in area over commercial synthesis. We have found 2-3X improvement in area and routing resource in C language applications tested. In addition, since most of the elements of the design are pre-placed and pre-routed, compilation time (from C to bit stream) is on the order of minutes rather than the hour(s) for traditional synthesis followed by Auto Place and Route.

The organization of the remainder of the paper is as follows. The next section reviews related work. Then we discuss the CLAy architecture. Next we describe the "module generator" Modgen tool, that enables a hardware designer to build parameterized macros, where the parameters can be bit length, pitch, and shape. Elements of the module library available to the compiler and other hardware designers are presented. The next section describes the synthesis strategy of the Malleable Architecture Generator (MARGE) compiler. The compiler creates custom "instructions" and maps those instructions to instantiations of macros in the module generator library. The final section summarizes our approach.

[1] In contrast, random control logic is more common in glue logic applications, for which coarse grained FPGAs are sometimes more versatile.

2 Related Work

Pre-placed, pre-routed macro libraries are commonplace tools provided by FPGA vendors. Our innovation is in the use of parameterized macro generators, and in the use of a macro definition language that allows parameterization of placement and routing choices. Related work in the latter area is the Logic Description Generator (LDG) [3], that was used on Splash-1 to create pre-placed, pre-routed systolic structures on Xilinx 3090 FPGAs. LDG was embedded in Lisp and was purely textual. In contrast the Modgen system has both a C-language-like input as well as graphical input. Modgen also has an associated simulation environment so that module generators can be developed and tested very rapidly, a capability that was not available with LDG.

Other C-to-hardware synthesis environments include the Prism system [1] and its successor Prism-II [12]. Luk and Page [8] have also developed a system to compile occam into FPGAs. Our approach differs from these efforts in that our programming model is a data parallel one in which the programmer controls the bit size of data and operations; that our synthesis is directed towards hybrid conventional processor/FPGA array environments; and that we specifically target highly optimized bit-slice organized macros.

3 The CLAy Architecture

The CLAy technology [11] employs a patented, RAM-based, regular architecture. Timing is regular and predictable due to the symmetrical architecture. An external clock can be used to synchronize the entire device or individual columns of cells, resulting in minimal skew.

The CLAy logic cell (up to 3 inputs, 2 outputs) implements a set of logic functions stemming from forty-four cell states including simple functions like NOR, AND, NAND, OR, XOR, INV, MUX, FlipFlop and complex functions using combinations of the above (e.g. Half-Adder, Registered Half-Adder, etc.).

The CLAy31 provides 3136 logic cells with an average usable gate count of approximately 5k gates at 30% utilization (the CLAy10 provides 1024 logic cells). The actual gate count is typically higher for compute-intensive designs. For example, a 12-bit, 26-tap low-pass filter constructed automatically using the module generation system yields a 13k gate design that utilizes 79% of the array.

A unique feature is its partial as well as full reconfigurability by cell. The device can dynamically reconfigure any given section without affecting the operation of other sections. The CLAy31 can be fully reconfigured in approximately 0.6 milliseconds, equivalent to a reconfiguration rate of 8 million gates per second.

The CLAy technology is also packaged in a Field Configurable Multi Chip Module (FCM). An FCM contains four CLAy 31 "tiles" arranged in a 2×2 array. The FCM features over 20,000 useable gates and 432 user I/O. It is packaged in a 625 pin Ball Grid Array with 448 inter-module connections per device, greatly reducing inter-tile routing delay. The Napa array consists of one or more FCMs plus SRAM on a PCI bus [5].

4 Module Generator

At the heart of our synthesis capability is the module generation system Modgen developed by Charle' Rupp for National Semiconductor. The module generation system enables a designer to build a library of parameterized generators that allow the creation of a specific function with any number of bits (subject to the size limitations of the chip) with tradeoffs between speed and area.

The library of generators encapsulate construction algorithms specific to the architecture which reflect design and layout complexity captured from an expert designer for the technology, thereby maximizing functional density and performance. The actual generators exist in a C-like language called D4[9] and are easily written and tested once a specific function has been identified and prototyped. At the present time, the library of generators contain most of the commonly used compute functions used in datapath design.

4.1 Module Library

Figure 1 shows a partial list of generators available in the library[10].

Specifically, for the CLAy technology, Reed-Muller logic[7] has been highly leveraged for most of the compute-oriented functions as it offers an efficient alternative given the structure of the core cells in the array (XOR-based). Given the combination of this technique along with the expert place and route techniques imparted upon the generators, the macro functions result in extremely dense and performance-intensive manifestations for the technology.

As an example of quantitative comparison between functions generated using the module generators and functions generated using a combination of synthesis and automatic place and route tools, refer to Figure 2. The figure compares area and delay for several representative functions. Area is expressed in cells. Delay has been normalized, with the Modgen delay for each function as the unit delay. The figure shows that the synthesis designs occupy between 2.6 - 7.9 times as many cells as the equivalent Modgen designs. The synthesis designs have 1.6 - 4.3 times the delay of the Modgen designs.

Arithmetic	Control	Signal Processing	Memory
ALUs	Comparators	FIR Filters	Register Banks
Absolute Value	Encoders	Lin. Seq. Gen.	ROM
Adders	Decoders	CRC detect/gen	EEROM
Decrementers	Hi/Lo/True/Comp blocks	Gray Code converters	SRAM
Incrementers	Mask generators		
Counters	Multiplexors		
Shift Operations	Tristate Buses		
Boolean Funcs.			
Multipliers			
Rotators			

Figure 1: Module Generator Library

Function	Synth./APR	Modgen
8-bit Add	144 cells/2.75	56 cells/1
14-bit ADD	420 cells/4.3	98 cells/1
8x8 Parallel Mult.	2600 cells/1.6	330 cells/1
9x11 Parallel Mult.	2800 cells/1.8	550 cells /1

Figure 2: Area/Delay Comparison

The parameterized generators not only eliminate the need to maintain large predefined macro libraries, but more significantly accommodate the generation of functions with arbitrary bit length. Rather than using functions divisible by 4-bits (as offered in most coarse-grain architecture systems) and wasting the unused bit logic, functions of specific bit-lengths can be invoked. This results from the bit slice techniques used in the generators. Given a large number of functions using non-standard bit sizes, significant area can be saved.

Additional flexibilities exist in the generators to allow the designer to make area and speed tradeoffs (algorithm selection) as well as physical options to ease integration with other functions. For example, different multiplier algorithms allow speed area tradeoffs (e.g. Ripple Carry v.s. Carry Save). Examples of physical options are output spacing and control line placement. Output spacing (referred to as the "pitch" parameter) is important when abutting to other modules during final integration. If a module is driving another module with input pins residing on a pitch of 2, selecting a pitch 2 output spacing allows perfect abuttment with no routing overhead (thereby minimizing area and delay). Strategic physical placement of control lines also serves to minimize routing during the integration stage.

Figure 3 shows an example fragment of D4 code for an adder function. Although the code displayed here is purposely incomplete (to simplify the example), it illustrates the use of parameters (width, pitch, speed, options).

We have also gained efficiency by using generators for multiplexing structures (for control). Multiplexing three or less signals is generally achieved using 2:1 multiplex combinations. Multiplexing more than three signals becomes more efficient using a tristate structure, which packs quite nicely in the CLAy array compared to other technologies.

These functions are represented by very simple generators shown in Figure 4 (2:1 mux bank and tristate buffer bank). We have also employed more complex tristate-multiplex structures, for example, 2-dimensional (n x m) tristate banks.

These module generators can be used manually to develop designs. In addition, the module library is targeted by the MARGE synthesis tool.

4.2 Automatic Module Expansion

The library described above consists of module generators. To be used in a design, each generator must be expanded or instantiated to be incorporated into a design just as a macro in a conventional high level language must be invoked in order for the macro body to be instantiated into the program.

In order to make the module generator system work automatically with the MARGE C language synthesis system, we have developed a tool that invokes a generator whenever a generator call is detected in a behavioral or structural netlist. The tool scans the design netlist to glean all module generator instances. A "variants" file is then constructed which contains the list of all macros to be generated along with parameter values extracted from the instantiation. The

```
// ADD2 generator
processor
co,sum.width=_gen_add2(.width,.pitch,.speed,.options,A.width,B.width)
// pitch = 1  (speed 10) or 2 (speed 10 or 30)
// speed 10 => ripple, 30 => AP
// options & 1 => A in on NORTH bus
// options & 2 => A in on SOUTH bus
// options & 4 => B in on NORTH bus, Pitch 2 only
// options & 8 => B in on SOUTH bus, Pitch 2 only
{
int i,yloc,xloc = 0,w = width;
wire c[w+1];
.
.
.
  // pitch 1, speed 10 case for width less than 8 bits
  if ((pitch == 1) && ((speed == 10) || (w < 7)) )
    { yloc = w - 1;  co = c[w];
      AT xloc,yloc; c[1],bs[0] = _add_HA1_1(A.0,B.0);
      if (w > 1) ROUTE c[1],"AEB",xloc+3,yloc;
      for (i = 1; i < w; i = i + 1)
      { yloc = yloc - 1;
        AT xloc,yloc; c[i+1],bs[i] = _add_FA1_1(iside,A.i,B.i,c[i]);
        if (i < w - 1) ROUTE c[i+1],"AB",xloc+4,yloc;
        ROUTE c[i],"BAEB",xloc+4,yloc;
      }
    }
.
.
.
}
```

Figure 3: Example Add Module Generator

```
! 2-1 mux bank
processor Q.width = _gen_MUXBNK(.width,.pitch,D0_.width,D1_.width,S)
{ int i;
  for (i = 0; i < width; i=i+1)
  { AT 0,(width-i-1)*pitch; Q.i = PMUX.1(D0_.i,D1_.i,S);
  }
}
}
! Tristate buffer bank
! _gen_TRIB
processor bus Q = _gen_TRIBNK(.width,.pitch,A.width,OE)
{ int i;
  for (i = 0; i < width; i=i+1)
  { AT 0,(width-i-1)*pitch; @,Q = PBUFZ.1(A.i,OE);
  }
}
```

Figure 4: Multiplexor and Tristate Generators

variants file is then passed to the module generation engine to automatically generate all necessary macros along with any design representations required. The generated macros are essentially treated as a design library used in subsequent integration, verification and bitstreaming steps.

An example of a generator instantiation using Verilog syntax appears below:

```
wire [7:0] sum,w1,w2;
.
// add two  8-bit words, put result in sum
add2_9   i15  (sum,cr,w1,w2);
```

The part i15 is an add2_9 component. By convention, this reference consists of two parts, first the name of the generator, in this case add2, followed by the bit length of the operands, in this case 9. The other parameters to the add2 macro generator include the name of the result, sum, the name of the carry, cr, and the names of the two inputs, w1 and w2.

The compiler computes bit lengths of operands and result, and constructs the instantiation name based on the combination of operation to be performed and desired bit length.

5 Malleable Architecture Generator (MARGE)

The C compiler framework and Malleable Architecture Generator (MARGE) have been described in detail elsewhere.[4] [5] [6] In this section we briefly summarize the compilation framework. Our synthesis tool, MARGE, translates high-level parallel C to configuration bit streams for field-programmable logic based computing systems. Examples of such custom computing platforms include the Splash and Napa parallel arrays. Our programming model is data parallel, and the C variant we use is the data-parallel bit C (dbC) language. dbC constructs allow the programmer to choose what operations are to occur on the reconfigurable array and to specify the bit accuracy of operations. MARGE creates an application-specific instruction set that encompasses the parallel operations of the dbC program, and generates the custom hardware components required to perform those parallel operations. Sequence control and non-parallel operations occur on the host processor that controls the array.

Following the dbC programming model, our synthesis system divides the code generation problem into two parts, control flow and datapath. A host program is generated that performs sequential operations and sends custom hardware instructions to the reconfigurable array or Execution Unit (EU), which is configured as a linear array of virtual processors, one or more to an FPGA.

MARGE processes intermediate code in which each operation is annotated by the bit lengths of the operands. Each basic block (sequence of straight line code) is mapped into a single custom instruction which contains all the operations and logic inherent in the block. The synthesis phase maps the operations comprising the instructions into register transfer level structural components and associated control logic. The register transfer level components are mapped to module generator instantiations. The generated netlist is then processed as described in Section 4 to yield a final configuration bit stream.

5.1 Intermediate Code to RTL

To create an EU consisting of structural components, MARGE first establishes an instruction set from the datapath (non-sequence-control) operations. Each basic block (straight line code) is the basis for a single EU instruction. The basic block, represented by a directed acyclic graph (DAG), is separated into levels, with independent operations at each level executed in the same clock cycle. Thus each EU instruction consists of a sequence of cycles. At each cycle, one or more operation is performed. During synthesis, individual operations are mapped to Register Transfer Language (RTL) structural components and are scheduled: each operation becomes associated with a module generator instance and, in addition, is assigned an instruction number and a "tick" or cycle number. The basic elements of the RTL model used by MARGE are registers, functional units, routers, and control logic.

- **Registers**: Registers serve as the basic data storage element. Each variable in a program is assigned a register, and additional registers may be allocated for intermediate storage of information wherever needed. The Register Bank Generators are invoked for register elements.

- **Functional Units** (operators): Functional units are blocks that perform basic operations on data. Some examples of arithmetic and logical operators are adders, subtractors, comparators, bitwise ANDs and bitwise ORs. The Arithmetic Generators are used for functional unit elements.

- **Routing Control**: Data must be routed from registers to functional units to undergo some operation. In most cases these operations will result in some output value that must be routed back to a register for storage. In some simple cases the source of data may simply be hard-wired to the destination. The basic components used for more complicated routing are multiplexors and tri-state busses. Multiplexors are used when a destination has different data sources on different instructions: it will select between the possible sources during the appropriate instructions. When the number of input sources becomes sufficiently large it may be more efficient to replace the multiplexors with banks of tri-state gates. With the CLAy technology, tri-states become the more efficient option for three or more input alternatives. Control Generators are used for routing control.

- **Control Logic**: Basic logic gates (ANDs, ORs and NOTs) must be used to generate control signals to enable registers to latch data only on certain cycles of instructions. Similar control signals are needed to control routing circuitry to switch between different multiplexor inputs at different times.

The above components are controlled by an instruction decoder and a multitick decoder. These decoders respond to control signals from the host and direct the processing circuitry performing computations on the Execution Unit to perform the indicated set of operations at the indicated time.

By way of example consider the following four operation sequence[2]:

```
1) A=B+C; (operation 6, tick 2)
2) A=B+D; (operation 4, tick 0)
3) A=D-C; (operation 7, tick 3)
4) A=A+B; (operation 3, tick 1)
```

Since there are four variables in the program, four registers are needed (see Figure 5.) There are two operators used in the program: a subtractor is needed for instruction 3, and an adder is needed for instructions 1,2 and 4. The adder input is taken from four different sources on different instructions. The operand B is used in all addition operations, so the first input of the adder may be hardwired to the output of register B. However, the second input to the adder requires different sources on different instructions. The second input to the adder must be fed from the output of a tri-state router whose inputs are hardwired to be the outputs of registers A,C and D. The subtractor inputs are hard wired to be taken from the outputs of registers D and C. Since the source of the data being written to register A varies, a two input multiplexor is needed to route either the output of the adder or subtractor to its input.

The combination of signals from the instruction decoder and the multitick decoder creates a unique control signal that can uniquely identify every clock cycle of every instruction. These signals can be used to control multiplexor input selectors and to assert register enable inputs. Figure 6 illustrates the use of these signals in the context of the above example.

Since Register A is the destination of data on all four instructions, an enable signal must be generated to allow it to latch new data on all four instructions on the appropriate multitick cycles. The appropriate signals must also be generated to select the proper inputs of the multiplexor feeding register A. Likewise, the tristate enable signal must be created so that register C is routed to the adder on the first instruction, register D is routed to the adder on the second instruction, and register A is routed to the adder on the fourth instruction, all on the appropriate multitick cycles.

The example above maps to a relatively simple hardware description composed of basic functional building blocks. These building blocks are obtained from the Modgen module library, so that the Execution Unit operations are expressed in terms of optimized macro blocks.

The example above illustrates the basic methodology. MARGE also performs various optimizations:

- **Function unit reuse**: Function units are reused between operations whenever possible. It is possible to reuse a function unit if it is of the appropriate size and functionality and is not being used in a different operation on the clock cycle currently being scheduled.

- **Commutativity optimization**: Operands to a binary function may be swapped to reduce mux/tristate control logic. In the fourth instruction of the above example, MARGE swaps the operands of the commutative add operation so that the left input to the adder can be hardwired to B. Without the transformation, a mux would have been generated for the left input.

[2]Note that this is not a semantically meaningful sequence of operations. A more realistic scenario is that the four instructions are separated by other instructions, possibly including control flow. The example is simplified for clarity.

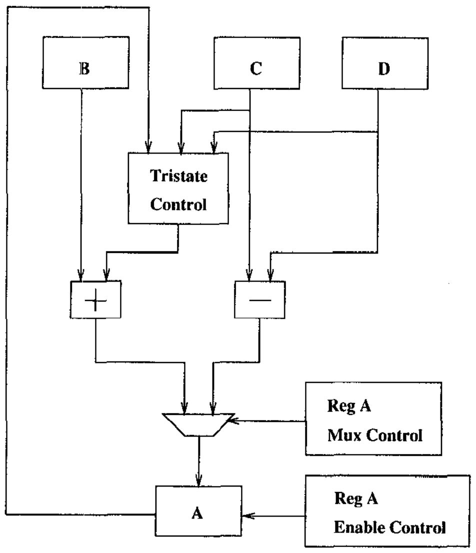

Figure 5: Structural Hardware Description from RTL methodology

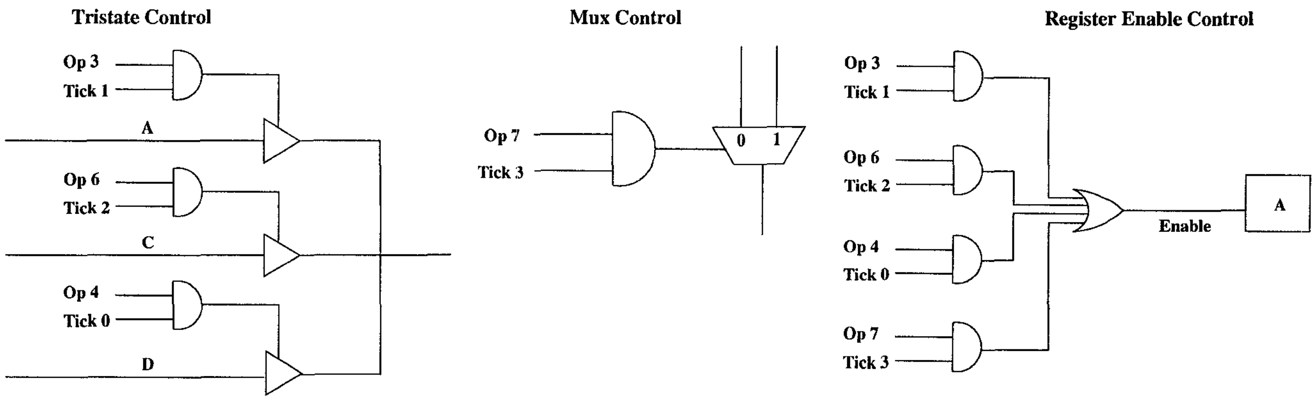

Figure 6: Data routing and control structures

Program	Synthesis/APR	Modgen
XCorr	1444 cells	570 cells
DNA	2703 cells	1800 cells
XCorr	153 nets	108 nets
DNA	403 nets	294 nets

Figure 7: Synthesis vs. Modgen on Two Applications

The MARGE synthesis module can also handle bit insertion and extraction operations, in which a portion of a register is stored or a range of bits from a register is extracted. These operations can be expressed at the high-level in dbC and map directly into wires (plus mux and tristate control if required) in the hardware.

Figure 7 shows comparative results from two programs XCorr, a bit stream cross correlation program, and DNA, a DNA sequence match program. Both programs use systolic algorithms. XCorr performs bit-oriented operations and sums into a 16-bit counter. DNA uses operands of 2- and 4- bits, where the 2-bit operands are used in compares, and the 4-bit operands are used in adds. The gate-level structural code emitted by the compiler was processed by two different sets of tools. The column labelled Synthesis/APR shows the result of processing the structural code with the Synopsys Design Compiler and then using CLAy technology specific tools to generate the bit streams. THe column labelled Modgen show the result of mapping MARGE's structural output to pre-placed, pre-routed macros from the CLAy Module Library. As the table shows, there is a dramatic reduction in the number of core cells used by the Modgen versions over the synthesis versions, with the synthesis versions taking up to 2.5 times as many cells. In addition, the number of nets, an indication of routing resources consumed, are also reduced in the Modgen versions.

6 CONCLUSIONS AND FUTURE WORK

We have described an automatic synthesis system that maps high level data parallel C to a library of hardware module generators. In order to make configurable computing systems accessible to the programmer, we have started with a high level procedural language rather than a hardware description language. We have obtained efficiency over traditional synthesis by targeting a library of pre-placed, pre-routed macro generators for a fine grained FPGA architecture. The macro generators have been designed by expert engineers to be arbitrarily expandable in bit width and to fit well with each other. Our techniques show significant area and delay improvements for Modgen elements over equivalent functions compiled through traditional synthesis and APR. In addition, a dbC application also demonstrates substantial improvement through MARGE/Modgen over traditional synthesis of the same application through MARGE/VHDL-synthesis.

We are currently developing an efficient datapath generator to be used on a new FPGA array (RSP Adaptive Logic Processor[2]) enhanced specifically for compute-intensive pipeline segments. The datapath generator will leverage rule-based techniques to physically integrate datapath macros that have been auto generated from set of generators enhanced for the new array. We expect to see significant improvement in density and performance due to the enhanced array and the new integration techniques. MARGE is being enhanced to support this new effort.

7 ACKNOWLEDGEMENTS

This work was supported by National Semiconductor Corp., DARPA through Contract DAB63-94-C-0085 to NSC and the David Sarnoff Research Center.

References

[1] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, March 1993.

[2] National Semiconductor Corporation. Reconfigurable signal processor. *http://splish.ee.byu.edu/arpa/arpa.html*, 1996.

[3] M. Gokhale, A. Kopser, S. Lucas, and R. Minnich. The logic description generator. *Conference on Application Specific Array Processors*, September 1990.

[4] M. B. Gokhale, J. Kaba, and J. A. Marks. Malleable architecture generator for fpga computing. *SPIE Conference on High Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, November 1996.

[5] M. B. Gokhale and A. Marks. Automatic synthesis of parallel programs targeted to dynamically reconfigurable logic arrays. In *Proceedings*

of the 1995 International Workshop on Field-Programmable Logic and Applications, Oxford, England, pages 399–408, September 1995.

[6] M. B. Gokhale and B. Schott. A data-parallel programming model. In *Splash-2: FPGAs in a Custom Computing Machine*, pages 76–78. IEEE Press, 1996.

[7] D. Green. *Modern Logic Design.* Addison-Wesley, 1986.

[8] I. Page and W. Luk. Compiling occam into FPGAs. In *FPGAs. International Workshop on Field Programmable Logic and Applications*, pages 271–283, Oxford, UK, September 1991.

[9] Charle' R. Rupp. D4 language reference guide and specification.

[10] Charle' R. Rupp. D4 user's guide.

[11] National Semiconductor. Configurable logic array data sheet. 1993.

[12] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II compiler and architecture. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16, Napa, California, April 1993.