DP_Gen: A Datapath Generator for Multiple-FPGA Applications†

Wen-Jong Fang[1], Allen C.-H. Wu[1], Ti-Yen Yen[2], and Tsair-Chin Lin[2]

[1]Department of Computer Science, Tsing Hua University
Hsinchu, Taiwan, 300, Republic of China

[2]Quickturn Design Systems, Inc., 440 Clyde Avenue,
Mountain View, California, 94043, U.S.A

Abstract

This paper presents a datapath generator for multiple-FPGA applications. This datapath generator is able to generate complex datapath designs described in HDLs. Our datapath generator uses a novel synthesis and partitioning approach which bridges the gap between RTL/logic synthesis and physical partitioning to fully exploit design structural hierarchy for multiple-FPGA implementations. Experiments on a number of benchmarking circuits and industry designs demonstrate that the generator can effectively and efficiently produce high-density multiple-FPGA datapaths.

1 Introduction

Because of their low manufacturing time and cost, Field Programmable Gate Arrays (FPGAs) have become the most popular Application-Specific Integrated Circuit (ASIC) for fast system prototyping. In addition, the development of reconfigurable hardwares by integrating FPGAs and Field Programmable Interconnect Chips (FPICs) has become the new trend in fast-prototyping and computation-intensive applications [1, 2, 3, 4, 5].

Most of computation-intensive applications contain a variety of data-processing elements that can be cast as datapaths. Since many datapaths are too large to fit into a single FPGA chip, multiple FPGA chips are required to implement such datapath designs. In general, the commonly used design flow to map datapaths onto a multiple-FPGA implementation consists of three phases. In the first phase, a synthesizer is used to transform the HDL description of a datapath into an RTL design. In the second phase, the RTL design is converting into a flattened CLB netlist by performing a series of logic optimization and technology mapping procedures. In the final phase, a partitioner is used to partition the CLB netlist into FPGA chips.

One of the crucial tasks for multiple-FPGA implementations is to partition a design onto a set of FPGAs. The problem of FPGA-based partitioning is quite different from the classical ASIC partitioning problem. FPGA chips have fixed and limited amounts of logic units (CLBs) and I/O pins. Typically, mapping a design onto a set of FPGA chips is predominately constrainted by I/O-pin limitations. This often results in FPGA partitions with very low logic utilizations. In the past several years, many partitioning approaches and algorithms [6, 7, 8, 9, 10, 11, 12] have been proposed to solve the multiple FPGA partitioning problem. However, all of the above approaches perform partitioning on flattened circuit-level netlists, which do not take into account design-hierarchy information. In a recent study [13], Schmit et al. experimented multiple FPGA partitioning at behavioral and structural levels. They have two interesting observations. Firstly, the best behavioral partitions do not always correspond to the best structural partitions. Secondly, during structural partitioning, the IO limitation can be reduced if the partitioner is capable of decomposing and placing portions of structural components such as multiplexors and controllers into different FPGA partitions. In addition, Isshiki and Dai [14] proposed a high-level bit-serial datapath synthesis method for multi-FPGA systems. This method aims to design datapaths using bit-serial circuits so that the partition quality is no longer dominated by the IO resource of the FPGA.

In this paper, we present a datapath generator for multiple-FPGA applications. This datapath generator is able to generate complex datapath designs described in HDLs. Moreover, the datapath generator uses a new synthesis and partitioning approach which bridges the gap between RTL/logic synthesis and physical partitioning by finely tuning logic implementations suited for multiple-FPGA implementations. Experiments on a set of benchmarking circuits and industry designs are reported. The results demonstrate that the generator can effectively and efficiently produce multiple-FPGA datapaths with high logic utilization.

The rest of paper is organized as follows. Section 2 gives an overview of the datapath generator. Section 3 presents the datapath generation method. In Section 4, we present the experimental results. Finally, Section 5 provides concluding remarks.

†Supported by the National Science Council of R.O.C. under contracts No. NSC-85-2215-E-007-011 and NSC 86-2221-E-007-047

ASP-DAC '97
0-89791-851-7/$5.00 ©1997 IEEE

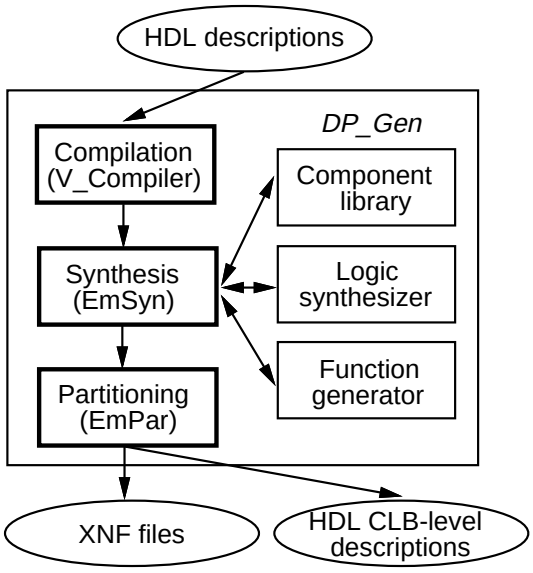

Figure 1: The system block diagram.

2 Overview

Figure 1 depicts the system block diagram of the datapath generator *DP_Gen* which consists of three major components: a Verilog compiler (*V_Compiler*), an RTL synthesizer (*EmSyn*), and a partitioner (*EmPar*). *EmSyn* interfaces to a set of logic minimization/technology mapping procedures, a component library, and a function generator. The input to the generator is a Verilog description of the design. *V_Compiler* performs HDL compilation and converts the Verilog design description into an intermediate design format. *EmSyn* first performs RTL synthesis to generate a structural design. Then, *EmSyn* invokes logic minimization and technology mapping procedures to convert the structural design into a CLB-based design. The component library provides the synthesizer with a set of generic RTL components, such as adders and subtractors, to support the RTL synthesis. In addition, the function generator can dynamically generate a set of bit-level logic sub-functions for bit-sliced components. If the generated datapath can not fit into a single FPGA chip, then a partitioner *EmPar* is used to decompose it into multiple-FPGA chips. Finally, the module generator outputs both an XNF and Verilog description of the datapath.

3 Datapath generation

The datapath generation consists of two phases: (1) datapath synthesis and (2) datapath partitioning. In the following sections, we first describe the datapath synthesis method. Then, we present the datapath partitioning approach.

```
module MUX2(o,i1,i2,sel);
parameter BIT_WIDTH=4;
  output [1:BIT_WIDTH] o;
  input [1:BIT_WIDTH] i1,i2;
  input sel;

  reg [1:BIT_WIDTH] o;

 always
   case(sel)
     1'b0: o = i1;
     1'b1: o = i2;
   endcase
endmodule
```

(a)

```
module MUX2(o,i1,i2,sel);
parameter BIT_WIDTH=4;
  output [1:BIT_WIDTH] o;
  input [1:BIT_WIDTH] i1,i2;
  input sel;

  assign o[1] = ((sel & i1[1]) | (~sel & i2[1]));
  assign o[2] = ((sel & i1[2]) | (~sel & i2[2]));
  assign o[3] = ((sel & i1[3]) | (~sel & i2[3]));
  assign o[4] = ((sel & i1[4]) | (~sel & i2[4]));
endmodule
```

(b)

Figure 2: The Verilog descriptions of a 4-bit 2-to-1 multiplexer: (a) behavioral-level, (b) logic-level.

3.1 Datapath synthesis

The datapath synthesis consists of three steps: (1) HDL compilation, (2) RTL synthesis, and (3) logic synthesis. The Verilog compiler *V_Compiler* first transforms a Verilog RTL description of the design into an intermediate design format. The RTL design can be described at behavioral or logic level. For example, Figures 2(a) and (b) show the behavioral-level and logic-level descriptions of a 4-bit 2-to-1 multiplexer. For an RTL behavioral description, the synthesizer will perform unit selection and unit/storage/interconnect binding, and output a structural design. The structural design consists of a set of interconnected regularly-structured functional units represented by a hierarchical function-tree in the Berkeley Logic Interchange Format (BLIF) and EQN (Boolean equation) format. For a logic-level description, the synthesizer will directly convert the design into a function-tree in BLIF and EQN formats. During logic synthesis, the synthesizer *EmSyn* invokes the logic minimizer and technology mapper [15] to generate CLB-based netlists of datapaths.

3.2 Datapath partitioning

When a datapath is too big to fit into a single chip, then it needs to be partitioned into multiple chips. One way is to apply an existing traditional circuit-level partitioning algorithm such as the RFM [16] to decompose a large CLB netlist into a set of subnetlists. However, in certain common cases, the RFM method produces partitions with high I/O-pin utilization but low logic utilization. When datapaths contain a set of multi-bit datapath components, the I/O limitation of the chip becomes the bottleneck and a high logic-utilization partition can not be achieved using the traditional circuit-level partitioning method.

In our datapath partitioning, we use a new RTL partitioning method [17] to improve the I/O-pin and logic utilizations of FPGAs. The main objective of the RTL partitioning method is to fully exploit the design structural hierarchy and to allow decomposing portions of the data-processing components into different FPGA partitions. The RTL partitioning method consists of two phases: (1) function-tree construction and

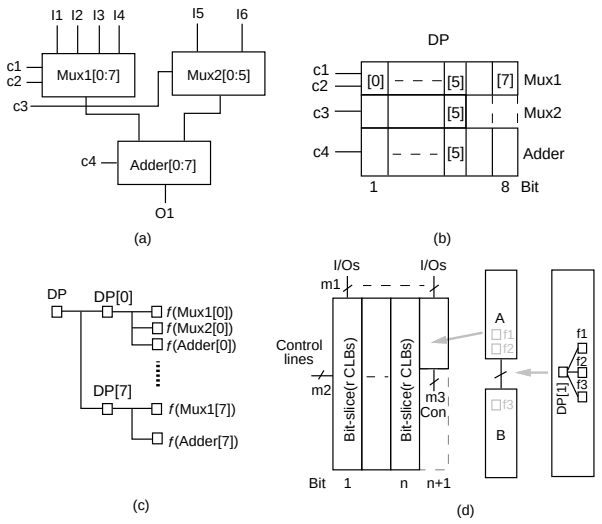

Figure 3: Functional structuring and partitioning: (a) an RT example, (b) the topological floorplan of the datapath, (c) the function-tree, (d) functional partitioning.

(2) functional partitioning.

3.2.1 Function-tree construction

Function-tree construction consists of three steps: (1) function decomposition, (2) function restructuring, and (3) CLB and IO-pin estimations.

In the first step, the generator invokes the *function generator* to decompose the functionalities of the RT components into a set of sub-functions. The logic function of a datapath component is decomposed into a set of bit-level sub-functions. Each sub-function represents one-bit of the component. A hierarchical function-tree is constructed by decomposing the functionality of the design in a top-down fashion.

In the second step, a hierarchical function-tree is reconstructed into a bit-level function-tree by performing bit-alignment and topological placement of the datapath components. A datapath may contain components with varying bit widths. For such an irregularly-shaped datapath, the components are aligned according to their connectivities. For example, in Figure 3(a), the datapath contains an 8-bit adder, an 8-bit multiplexer $Mux1$, and a 6-bit multiplexer $Mux2$, in which $Mux2$ is connected to the least-significant 6-bit of the *Adder*. The topological floorplan of the datapath is shown in Figure 3(b). According to the topological floorplan of the datapath, the first bit of the datapath contains three one-bit logic functions of $Adder[0]$, $Mux1[0]$, and $Mux2[0]$, as shown in Figure 3(c). On the other hand, the eighth bit of the datapath contains only two one-bit logic functions of $Adder[7]$ and $Mux1[7]$.

In the third step, we compute the required CLBs and I/O pins for each node of the function-tree. To obtain such information, we first perform FPGA synthesis to generate CLB netlists for the leaf nodes of the function-tree. For example, for the leaf-node of $f(Mux1[0])$ in Figure 3(c), we can obtain its CLB netlist by invoking the logic minimizer and technology mapper [15] with logic function of $f(Mux1[0])$. After generating the CLB netlists for all the leaf nodes, we can generate the CLB netlists for intermediate nodes of the function-tree by applying the collapsing technique described in [15]. Consequently, the required CLBs and I/O pins of nodes in the function-tree can be determined. Furthermore, the number of interconnections between two nodes can be computed by matching the I/O pins of these two nodes. If the design can be fit into a single FPGA chip; that is, the number of CLBs and IO pins of the *Root* node satisfies the CLB and IO-pin constraints of the chip, then the datapath generation terminates. Otherwise, a functional partitioning procedure will be invoked which will be discussed in the following section.

Let G be an RTL netlist. DP denote the datapath component set. $f(DP)$ denote the logic-function sets of the datapath components. In addition, CLB and IOP represent the CLB and IO-pin constraints of the FPGA chip. The pseudo code of the function-tree-construction procedure is listed as follows.

ALG. Function_Tree_Construction(G, DP)\{
  $f(DP) = Function\_Generation(DP)$;
  $Bit\_Alignment(G, DP)$;
  $T = Bit\_Level\_Function\_Tree(f(DP))$;
  $CLB\_IO\_Estimation(T)$;
  if $(Clb(Root(T)) \leq CLB$
    and $IO(Root(T)) \leq IOP)$ **then**
    $Datapath = Netlist(Root(T))$;
  else
    $Functional\_Partitioning(T)$;
\}

Procedure *Function_Generation* generates bit-level logic functions for datapaths. Procedure *Bit_Alignment* performs bit-alignment of the datapath components. Procedure *Bit_Level_Function_Tree* builds up the function-tree according to the bit-alignment of the datapath components. Procedure *CLB_IO_Estimation* invokes logic minimization and technology mapping algorithms to convert the logic functions into CLB-based designs. If the design can be fit into a single chip, then the CLB netlist at the root of the function-tree is assigned to a chip *Datapath*. Otherwise, the *Functional_Partitioning* procedure will be invoked to decompose the design into multiple chips, which will be discussed in the next section.

Complexity analysis: Let BW be the average bit widths of all of the components in DP, n the number of nodes in the netlist, m the number of edges in the netlist. The *Function Generation*, *Bit_Alignment*, and *Bit_Level_Function_Tree* procedures take $O(BW \times n)$, $O(m + n)$, and $O(BW \times n)$ time, respectively. The *CLB_IO_Estimation* procedure performs logic minimization and technology mapping for each node in the

function-tree. The computational complexity of this procedure is dependent upon the logic minimization and technology mapping algorithms used.

3.2.2 Functional partitioning

During functional partitioning, we use a bit-slice of the datapath as the basic unit and pack the bit slices into FPGAs from the least significant bit (LSB) to the most significant bit (MSB) in sequential order. The objective is to maximize the CLB-utilization of the FPGA chips subject to satisfying the CLB-capacity and I/O pin constraints of the chips.

Considering one bit-slice in Figure 3(d), it contains r CLBs, m_1 I/O pins, and m_2 control pins. Hence, by assigning one bit-slice into an FPGA, it uses $m_1 + m_2$ I/O pins. By packing n bit slices into one FPGA, it will use up to $r \times n$ CLBs and $(n \times m_1) + m_2$ I/O pins, as shown in Figure 3(d). Assume that we can not pack the $n+1$ bit-slice further because of the CLB-resource constraint. However, we may be able to pack a portion of one bit-slice into the FPGA to improve the CLB utilization. For instance, in Figure 3(d), the final partition consists of n bit slices with a portion of the $n+1$ bit-slice.

The function partitioning procedure consists of two steps: (1) initial bit-slice packing and (2) bit-level cell packing. In the first step, we determine the maximum number of bit slices which can fit into one FPGA. We use the bin-packing algorithm to cluster bit slices into FPGAs one at a time under the given CLB-capacity and I/O-pin constraints. After the initial packing, the number of unused CLBs and I/O pins of the FPGA is then computed.

In the second step, a bit-level cell packing procedure is used to improve the CLB utilization of the FPGA chip. Let CLB and PIN be the unused CLBs and I/O pins. The bit-level cell packing problem is to partition the logic functions of one bit-slice into two subsets such that the CLB-capacity of one subset is maximized subject to satisfying the CLB and PIN constraints, as shown in Figure 3(d). We first use the bin-packing algorithm to pack logic functions of one bit-slice into clusters and then perform iterative improvement using a pairwise exchange procedure. The logic functions are packed one at a time based on a priority function.

Let C denote a set of chip used. $f(BS)$ and $f(BT)$ represent a set of logic functions of bit slices and portions of one bit-slice, respectively. $T_{bit} = \{V, E\}$ denotes a sub-tree represented the logic functions of one bit-slice, where V is a set of vertices representing bit-sliced logic functions, $V = \{v_i \mid i = 1..n\}$, and E is a set of edges representing the connections between logic functions, $E = \{e_{ij} \mid v_i, v_j \in V\}$. V_1 and V_2 represent two subsets of vertices. $Clb(v_i)$ denotes the number of CLBs in v_i and $w(e_{ij})$ denotes the number of connections between v_i and v_j. $conn_k(v_i)$ denotes the number of connections between v_i and the vertices in subset V_k. $c(v_i)$ denotes the interconnect cost of v_i. The pseudo code of the functional partitioning algorithm is listed as follows:

```
ALG. Function_Partitioning(T, C){
  C = φ; i = 1;
  while (T ≠ φ){
    {f(BS), CLB, PIN} = Bit_Slice_Packing(T);
    f(BT) = Cell_Packing(T_bit, CLB, PIN);
    c_i ← f(BS) ∪ f(BT);
    T = T - (f(BS) ∪ f(BT));
    C = C ∪ c_i; i++;
  }
}

  PROC. Cell_Packing(T, CLB, PIN){
    V_1 = φ; V_2 = V;
    Priority_Function(V_1,V_2);
    v_i = Best_Fit(V_2,CLB,PIN);
    while (v_i ≠ φ){
      V_1 = V_1 ∪ v_i;
      V_2 = V_2 - v_i;
      CLB = CLB - Clb(v_i);
      Priority_Function(V_1,V_2);
      v_i = Best_Fit(V_2,CLB,PIN);
    }
    Pairwise_Exchange(V_1, V_2);
    Return(V_1);
  }

  PROC. Priority_Function(V_1,V_2){
    for (all v_i∈V_2){
      V_2 = V_2 - {v_i};
      conn_1(v_i) = ∑w(e_ij), for all v_j ∈ V_1;
      conn_2(v_i) = ∑w(e_ij), for all v_j ∈ V_2;
      c(v_i) = conn_2(v_i) - conn_1(v_i);
      V_2 = V_2 + {v_i};
    }
    /*Normalize the interconnect cost*/
    min_c = minimum_c(v_i)(V_2);
    c'(v_i) = c(v_i) - min_c + 1;
    sc(v_i) = Clb(v_i) ÷ c'(v_i), for all v_i ∈ V_2;
  }
```

Procedure $Bit\_Slice\_Packing$ determines the maximum number of bit slices which can fit into one chip and returns the number of unused CLBS (CLB) and IO-pins (PIN) of the chip. Procedure $Cell\_Packing$ returns the portions of one bit-slice which can be packed into the chip. Procedure $Priority\_Function$ scores each cell ($sc(v_i)$) based on the ratio of its CLB-capacity ($Clb(v_i)$) and the interconnect cost ($c(v_i)$). $c(v_i)$ indicates the interconnect gain by moving v_i from subsets V_2 to V_1. When $c(v_i)$ is a negative value, it means that the cut-lines between V_1 and V_2 are reduced by $c(v_i)$ by moving v_i from subset V_2 to V_1. Normalization is done such that all interconnect costs have positive values. When a cell v_i has a large $sc(v_i)$ value means that more CLBs can be packed into FPGAs with consuming less I/O pins. Procedure $Best\_Fit$ first sorts the cells in V_2 in descending order according their $sc(v_i)$ scores. Then, the procedure searches the first cell v_i such that $(Clb(v_i) \leq CLB)$ and $(Interconnect\_Cost(V_1 \cup v_i, V2 - v_i) \leq PIN)$. If such a cell is found then return the cell. Otherwise, it returns an empty set ϕ. Finally, the algorithm performs a pairwise exchange procedure

($Pairwise\_Exchange(V_1, V_2)$) to iteratively improve the CLB utilization under the I/O pin constraint.

The *Functional_Partitioning* algorithm runs recursively to partition the datapath components (starting from the LSB slice) into FPGAs one chip at a time. The procedure terminates when all the components are assigned into FPGA chips.

Time Complexity. Procedure *Bit_Slice_Packing* takes $O(1)$ time. Let n be the number of logic functions of one bit-slice and m the average number of connections associated with each cell. Procedures *Priority_Function* and *Best_Fit* take $O(m \times n)$ and $O(nlogn)$ time, respectively. Hence, the *Functional_Partitioning* algorithm takes $O(n \times ((m \times n) + (nlogn)))$ time.

4 Experiments

We have implemented the datapath generator in the C programming language. Presently, the generator is embedded in an interactive multiple-FPGA synthesis and partitioning system (*ISyn*) which consists of approximately 150,000 lines of C code and runs on SUN and HP workstations.

We have tested our generator on two benchmarking circuits and two industry designs, as shown in Table 1. The first benchmark is an ALU from the Mano book [18]. The second one is the fifth-order elliptic filter which is extensively used in high-level synthesis. The bit-width of the two benchmarks is 32. *Industry I* and *II* are two industry designs, a datapath and a floating-point multiplier. We have targeted to two different technologies: the Xilinx 3000 series and the Xilinx 4000 series chips. Table 1 depicts the characteristics of the benchmarks, in which $\#IOs$, $\#CLBs$, $\#Eq.Gates$, $\#Pins$, and $\#Nets$ represent the number of IOs, CLBs, equivalent gate counts, pins, and nets of the designs.

We have compared the partitioning results produced by our approach and a traditional approach. In the traditional approach, we first used *ISyn* to generate a flattened CLB netlist. Then we applied the RFM algorithm to partition the flattened netlist into multiple-FPGA chips. Table 2 shows the comparative results in which *Chips*, *IO_U*, and *CLB_U* represent the number of partitions, the average I/O utilization, and the average CLB utilization, respectively. We have targeted to four different chips: (1) XC3090 with 144 IO pins and 320 CLBs, (2) XC3042 with 96 IO pins and 144 CLBs, (3) XC4010 with 160 IO pins and 400 CLBs, and (4) XC4005 with 112 IO pins and 196 CLBs. The results show that our approach produced partitions with lower IO-utilization (in average 69%) and higher CLB-utilization (in average 86%) compared to that produced by the RFM algorithm (in average 92% IO-utilization and 56% CLB-utilization). Only 3 out of 16 partitions (*Multiplier*, *IndustryI*, and *IndustryII* targeted to XC3090 chips, and *IndustryII* targeted to XC3042 chips), the RFM algorithm achieved the same CLB utilizations as that produced by our approach. Nevertheless, they consumed in average 15% more IO pins. This demonstrates that I/O limits are the bottleneck for CLB usage enhancement when the RFM algorithm was performed on flattened circuits. On the other hand, the CLB and I/O-pin utilizations are significantly improved when the functional structuring and partitioning approach was performed on hierarchical circuits.

Because each FPGA chip has only limited routing resources, a high CLB-utilization partition (e.g., $\geq$90%) may result in an unroutable design. We have tested the routability of partitions on some high CLB-utilization designs. For example, we used Xilinx *PPR* to perform place-and-route tasks on the elliptic filter design which has 96%, 88%, 99%, and 96% CLB-utilizations for XC4010, XC4005, XC3090, and XC3042 chips, respectively. The results show that both partitions targeted to XC4010 and XC4005 chips were routable even under the high CLB-utilization condition. On the other hand, both partitions targeted to XC3090 and XC3042 were unroutable. This indicates that the routing capability of a chip is dependent on the chip architecture.

5 Conclusions

In this paper, we have presented a datapath generator for multiple-FPGA implementations from RT netlists. We have tested our generator on a number of benchmarks and industry designs. Experimental results have demonstrated that our datapath generator is able to produce high-density multiple-FPGA datapaths.

Our generator is most applicable to low-speed applications, such as hardware emulation, due to its focus on the efficiency of CLB and IO-pin utilizations. Further study of timing issues would be beneficial for high-speed applications. We have also shown that the routing capability of chips depends on the chip architecture. In order to achieve viable partitioning solutions, further study is needed of the practicality of considering routability issues during the partitioning process.

Acknowledgment

The authors would like to thank Quickturn Design System Inc. and Dr. K. C. Chu for their support.

References

[1] M. Butts, J. Batcheller, and J. Varghese, "An Efficient Logic Emulation System," *Proceedings of ICCD92*, pp. 138-141, 1992.

[2] C. E. Cox and W. E. Blanz, "GANGLION- A Fast Field-Programmable Gate Array Implementation of a Connectionist Classifier," *IEEE Journal on Solid-State Circuits*, vol. 27, pp. 288-299, March 1992.

[3] P. K. Chan, M. Schlag, and M. Martin, "BORG:A Reconfigurable Prototyping Board Using Field-Programmable Gate Arrays," *in Proceedings of 1st International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, pp. 47-51, 1992.

[4] S. Walters, "Computer-Aided Prototyping for ASIC-Based Systems," *IEEE Design and Test of Computers*, pp. 4-10, June 1991.

Table 1: Characteristics of the benchmarking circuits.

Circuits	#IOs	#CLBs(A/B)	#Eq. Gates	#Pins(A/B)	#Nets(A/B)
Mano-ALU	47	849/694	5325	5321/6982	1275/1519
Elliptical Filter	117	2223/1549	13587	12857/14582	2739/3179
Industry I	46	1134/861	7151	6635/7826	1378/1798
Industry II	109	1752/1150	11050	9839/11590	1953/2370

A: XC3000, B: XC4000.

Table 2: Comparisons between our approach and RFM.

Circuits	Types	Ours			RFM		
		Chips	IO_U	CLB_U	Chips	IO_U	CLB_U
Mano-ALU	XC3090	3	.45	.88	7	.93	.38
Elliptic Filter	XC3090	7	.57	.99	13	.91	.53
Industry I	XC3090	4	.62	.89	4	.83	.89
Industry II	XC3090	7	.70	.78	7	.82	.78
Mano-ALU	XC3042	6	.63	.98	13	.92	.47
Elliptic Filter	XC3042	16	.72	.96	26	.91	.59
Industry I	XC3042	8	.85	.98	10	.85	.79
Industry II	XC3042	15	.81	.81	15	.92	.81
Mano-ALU	XC4010	2	.58	.87	3	.93	.58
Elliptic Filter	XC4010	4	.63	.96	12	.95	.32
Industry I	XC4010	3	.63	.72	4	.93	.54
Industry II	XC4010	5	.88	.58	8	.96	.36
Mano-ALU	XC4005	4	.60	.88	5	.92	.71
Elliptic Filter	XC4005	9	.70	.88	26	.96	.30
Industry I	XC4005	5	.81	.87	9	.98	.49
Industry II	XC4005	9	.87	.65	16	.98	.37
Average			.69	.86		.92	.56

[5] *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993, 1994, 1995, and 1996.

[6] C. Kring and A. R. Newton, "A Cell-Replicating Approach to Mincut-Based Circuit Partitioning," *Proceedings of ICCAD91*, pp. 2-5, 1991.

[7] G. Saucier, D. Brasen, and J. P. Hiol, "Partitioning with Cone Structures," *Proceedings of ICCAD93*, pp. 236-239, 1993.

[8] R. Kuznar, F. Brglez, and K. Kozminski, "Cost Minimization of Partitions into Multiple Devices," *Proceedings of the 30th DAC*, pp. 315-320, 1993.

[9] N.-C. Chou, L.-T. Liu, C.-K. Cheng, W.-J. Dai, and R. Lindelof, "Circuit Partitioning for Huge Logic Emulation Systems," *Proceedings of the 31st DAC*, pp. 244-249, 1994.

[10] N.-S. Woo and J. Kim, "An Efficient Method of Partitioning Circuits for Multiple-FPGA Implementation," *Proceedings of the 30th DAC*, pp. 202-207, 1993.

[11] D. J.-H. Huang and A. B. Kahng, "Multi-Way System Partitioning into a Single Type or Multiple Types of FPGAs," *Proceedings of 3rd International Symposium on FPGAs*, pp. 140-145, 1995.

[12] P. K. Chan, M. Schlag, and J. Y. Zien, "Spectral-Based Multi-Way FPGA Partitioning," *Proceedings of 3rd International Symposium on FPGAs*, pp. 133-139, 1995.

[13] H. Schmit, L. Arnstein, D. Thomas, and E. Lagnese, "Behavioral Synthesis for FPGA-based Computing," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines 1994*, pp. 125-131, 1994.

[14] T. Isshiki and W. W.-M. Dai, "High-Level Bit-Serial Datapath Synthesis for Multi-FPGA Systems," *Proceedings of 3rd International Symposium on FPGAs*, pp. 167-174, 1995.

[15] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithms for Table Look Up Architectures," *Proceedings of ICCAD91*, pp. 564-567, 1991.

[16] C. M. Fiduccia and R. M. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions," *Proceedings of 19th DAC*, pp. 175-181, 1982.

[17] W.-J. Fang and Allen C.-H. Wu, "A Hierarchical Functional Structuring and Partitioning Approach for Multiple-FPGA Implementations," *ICCAD96*.

[18] M. M. Mano, *Computer Engineering Hardware Design*, Prentice Hall Inc., 1988.