Check for updates

Module Generation of Complex Macros for Logic-Emulation Applications

Wen-Jong Fang[1], Allen C.-H. Wu[1], and Duan-Ping Chen[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, 94043-2232, U.S.A

Abstract

Logic emulation is a technique that uses dynamically reprogrammable systems for prototyping and design verification. Using an emulator, designers can realize designs through a software configuration process and perform real-time design verification before fabricating the chip into silicon. However, converting designs into an emulator involves the use of multi-phase design tasks, which is a very time-consuming process. Hence, shortening the Time-To-Emulation (TTE) is always the main concern for the logic-emulation design process. One approach to shorten the design processing time is to replace portions of the design with macro cells. This paper presents a module generator for logic-emulation applications, which is able to generate macro cells of arbitrarily complex functions described in High-level Descriptive Languages (HDLs). Furthermore, the module generator can effectively generate a multiple-FPGA macro for large macros which can not fit in a single FPGA chip. Experiments using the module generator for logic emulation are reported. The results demonstrate that the module generator can effectively and efficiently generate complex macros from their Register-Transfer-Level (RTL) description. In addition, the results also show that the design processing time is significantly shortened when the module generation method is incorporated into the logic-emulation design flow.

1 Introduction

Because of their reprogrammability, Field Programmable Gate Arrays (FPGAs) have become the most popular Application-Specific Integrated Circuits (ASICs) for rapid system prototyping. In addition, the development of reconfigurable systems by integrating FPGAs and Field Programmable Interconnect Chips (FPICs) has become the new trend in rapid prototyping and computation-intensive applications [1, 2, 3, 4, 5].

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
FPGA97, Monterey California USA
© 1997 ACM 0-89791-801-0/97/02 ..$3.50

Logic emulation is the first technique to emerge that uses dynamically reprogrammable systems for prototyping and design verification [1, 4, 6]. A logic emulator is a system consisting of hundreds and thousands of FPGAs and FPICs, which is able to realize designs through a software configuration procedure. A configured logic emulator is equivalent to the chip under design, and can be used for real-time design verification, software development, and prototyping before fabricating the chip into silicon. Using logic emulation, designers can run design verification almost six orders-of-magnitude faster than classical simulation [7]. In addition, designers are also able to execute complex system-level verification tasks such as running correctness tests, booting an operating system, and running application programs before tapeout. In recent years, many studies [4, 7, 8] have demonstrated that logic emulation is a very effective methodology for shortening the time-to-market of a design. Because of these advantages, logic emulation has been widely used as a key design verification methodology in many complex CPU, telecom, multimedia, and system design projects [4, 7, 8, 9].

Figure 1(a) depicts a typical design flow for logic-emulation applications [10]. A design is usually described as a mixed gate, logic, and Register-Transfer-Level (RTL) description in High-level Descriptive Languages (HDLs) such as VHDL and Verilog. In the first phase, a synthesizer is used to transform an HDL description into a gate-level netlist. In the second phase, a partitioner decomposes the gate-level netlist into a set of subnetlists such that each subnetlist can be realized using an FPGA chip. In the third phase, a system mapper first assigns the subnetlists into the FPGAs on the system board and then performs system routing to realize the interconnections between the FPGAs. In the fourth phase, a technology mapper converts the gate-level subnetlists into CLB subnetlists. In the fifth phase, a placer-and-router performs FPGA placement-and-routing for each subnetlist. Finally, the design is converted into a set of bit-stream files and downloaded into the target emulator.

Considering that a typical medium-sized design ranges between a hundred thousand and a half-million gates, and a large design may well over a million gates. Partitioning designs in such scales requires a tremen-

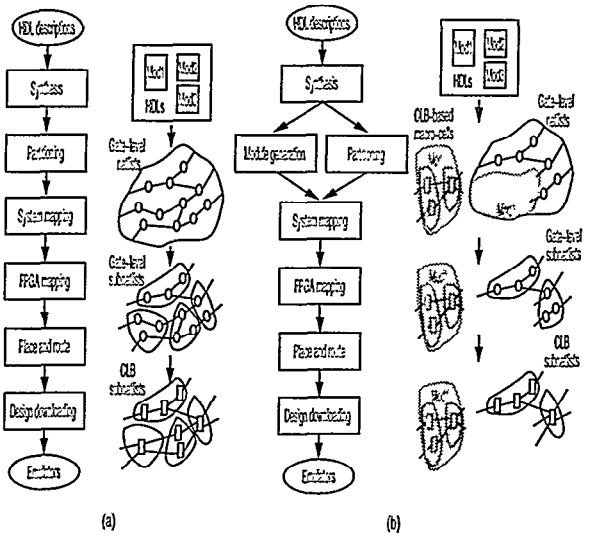

Figure 1: The design flow for logic-emulation applications.

dous computational effort which usually takes tens of hours. Moreover, in order to resolve the hold-time violation [10], a very complex timing analysis procedure has to be applied for path-delay analysis of the design before performing the partitioning. However, the number of paths that must be searched is enormous. To conduct path-delay analysis on such designs becomes an extremely time-consuming task. As a result, the above design process often suffers in a long design processing time, and how to shorten the time-to-emulation is always the top priority for the logic-emulation design process.

One possible approach to speeding-up the design processing time is to use a module generation approach, as shown in Figure 1(b). In this approach, some non-timing-critical portions of the design can be extracted and implemented as macro cells. Hence, we only need to apply the timing-analysis and partitioning tasks on the remainder of the design instead of the entire design. For example, in Figure 1(b), *Mod1* is extracted from the design and implemented as a macro cell. As a result, we only need to apply the timing analysis and partitioning tasks to the remaining circuits generated by *Mod2* and *Mod3*. By replacing portions of the design with macro cells, the computational complexity as well as the design processing time of the timing-analysis and partitioning tasks can be reduced.

In general, there are two commonly used approaches to implementing macro cells for logic-emulation applications. First, the macro cells can be developed manually by designers. However, this approach is a very time consuming process. Second, an existing macro library, such as the XACT macro library [11], can be used. While macro libraries provide a large number of basic components, such as adders, multiplexers, and counters, they don't provide macros for arbitrarily complex functions.

In this paper, we present a module generator for logic-emulation applications. This module generator is able to generate macro cells of arbitrarily complex functions described in HDLs. Moreover, the module generator can automatically provide a multiple-FPGA solution if a macro cell is too big to be realized by a single FPGA chip. Experiments using the module generator for logic emulation are reported. The results demonstrate that the module generator can effectively and efficiently produce complex macros. Furthermore, the design processing time is significantly shortened when the module generation method is incorporated into the logic-emulation design flow.

The rest of paper is organized as follows. Section 2 discusses the considerations of module generation for logic emulation. Section 3 presents the module generator. In section 4, we present the experimental results. Finally, section 5 gives concluding remarks.

2 Considerations of module generation for logic emulation

In this section, we discuss the characteristics of macros and the requirements of module generation for logic-emulation applications.

For logic-emulation applications, a pre-defined macro may contain a simple function or an arbitrarily complex function which can be described in HDLs at RT, logic, and gate levels. For example, a pre-defined macro can be a 32-bit 20-to-1 multiplexer, a custom-designed component with 5 arithmetic and logic functions, a 64-bit datapath, a multiplier, or even a stand-alone controller. As indicated in [8], gate-level descriptions of macros may not be available at the early design emulation stage. However, the RTL descriptions of macros are usually available for simulation use. In current practice, a functionally equivalent gate-level description must be created solely for emulation purposes. This will later be replaced by the actual gate-level description. Therefore, the module generator should be capable of transforming an RTL description of a macro into a gate-level description for emulation use. Another requirement stems from the fact that most emulation systems use commercial FPGA components to implement logic functions of the design. For example, Xilinx XC3090 and XC4013 chips are currently used in a variety of emulation systems. Hence, it is beneficial to the emulation design process if the module generator can produce a CLB-level description instead of a gate-level description.

Since an emulation-based macro may contain an arbitrarily complex function, a single FPGA chip may not be an adequate implementation. Thus, the module generator should be able to decompose a large macro into a set of FPGA chips. One way is to apply an existing traditional circuit-level partitioning algorithm such as the RFM [14] to decompose a large CLB netlist into a set of subnetlists. However, 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 and I/O pins. In certain common cases, the circuit-level partitioning method produces partitions with high I/O-pin utilization but low logic utilization. When macros contain a set of multi-bit data-processing components, the I/O limitation of the chip becomes the bottleneck and a

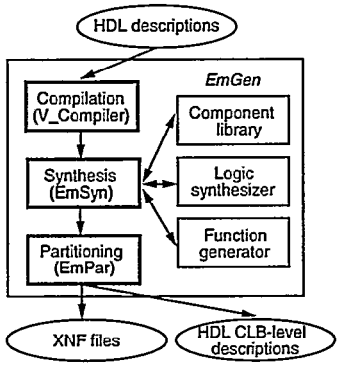

Figure 2: The system block diagram of the module generator *EmGen*.

high logic-utilization partitioning can not be achieved using the traditional circuit-level partitioning method. This is undesirable for emulation purpose because designs with low logic utilization will increase the emulation cost while high I/O utilization will decrease the ability to probe the design.

From the above discussion, we can conclude that an emulation-based module generator should be able to generate CLB-based macro cells from their RTL descriptions in HDLs. In addition, the module generator should also be able to decompose a large macro cell into a set of FPGA chips with high logic and low I/O-pin utilizations. In the following section, we will present an emulation-based module generator which provides all of the above features.

3 EmGen: an emulation-based module generator

Figure 2 depicts the system block diagram of the module generator *EmGen* 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 and a component library. 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 embedded in the logic synthesizer 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 macro can not fit into a single FPGA chip, then a partitioner *EmPar* is used to decompose the macro into multiple-FPGA chips. Finally, the module generator outputs both an XNF and Verilog description of the macro.

4 Module generation

In our module generation, we use a new RTL synthesis and partitioning method [16] to fully exploit the structural hierarchy of designs for multiple-FPGA implementations. The module generation consists of four steps: (1) synthesis, (2) component partitioning, (3) function-tree construction, and (4) functional partitioning. In the first step, the RTL description of a macro is coverted into a structural design. In the second step, the RTL components of the structural design are partitioned into two groups: bit-sliced and random-logic components. In the third step, a bit-level function-tree is constructed according to the structural characteristic of the design. In the final step, a functional partitioning method is applied to decompose the design into multiple FPGA chips.

4.1 Synthesis

In our approach, we use the FSMD (Finite State Machine with a Datapath) model [12] as our target architecture. We use the FSMD model to describe design behaviors on the register-transfer level. The Verilog compiler first transforms a Verilog RTL description of the design into an intermediate design format. The synthesizer performs unit selection, unit/storage/interconnection binding, control synthesis, and outputs a structural design. The structural design consists of a control unit (CU) and a datapath (DP). The control unit contains a set of random-logic functions for every datapath control signals. The datapath contains a set of interconnected RTL components including functional units, storage elements, and interconnect units. Each RTL component is represented as a set of logic functions in the Berkeley Logic Interchange Format (BLIF) and EQN (Boolean equation) format. During logic synthesis, the synthesizer invokes the logic minimizer and technology mapper [13] to generate CLB-based netlists of macros.

4.2 Component partitioning

The purpose of component partitioning is to determine the component style, bit-slice or random-logic, for each register-transfer component. The component style depends on the component type and the component's connectivity. First, components must be partitioned by type since some components, such as counters, registers and ALUs, are sliceable while others like, decoders and encoders are not. Second, small size components can be implemented in two ways. For example, a 2-bit ALU can be implemented as a random-logic or as a bit-sliced unit. The implementation decision for such a component depends on the component's connectivity. For example, if a component in question is strongly connected to other random-logic components, then a random-logic style may be more suitable for this component in order to reduce the interconnections between bit-sliced and random-logic modules.

A weighted and undirected hypergraph $G = (V, E)$ is formed from the RTL netlist. $V = \{v_i\}, i = 1..n$, denotes a set of components in the netlist, and $type(v_i)$ denotes the component type of v_i, *RL* (*Random_Logic*) or *BS* (*Bit_Slice*). $u_j(v_i)$, $j = 1..m$, denotes the ports of v_i, while $port(u_j(v_i))$ denotes the port type of $u_j(v_i)$, *control* port or *data* port. $E = \{e_{ij,kl}\}$ denotes a set of edges where $e_{ij,kl}$ is the edge between $u_j(v_i)$ and $u_l(v_k)$. In addition, $w(e_{ij,kl})$ denotes the weight of $e_{ij,kl}$ which

is the number of wires between $u_j(v_i)$ and $u_l(v_k)$. For example, in Figure 3(a), node v_1 has 4 ports where $p1(v_1)$, $p2(v_1)$, and $p4(v_1)$ are data ports, while $p3(v_1)$ is the control port. The number of wires between $p4$ of v_1 and $p2$ of v_2 is 6; i.e., $w\{e_{14,22}\}$=6.

Two linking costs, $C_{control}(v_i)$ and $C_{data}(v_i)$, are used to evaluate the connectivities between components. The linking costs of node v_i are computed as below:
$C_{control}(v_i) = \sum w(e_{ij,kl})$ where $port(u_l(v_k))$=*control* and $j = 1..m$, and
$C_{data}(v_i) = \sum w(e_{ij,kl})$ where $comp(v_k)$=BS and $port(u_l(v_k))$=*data*.

The linking cost $C_{control}(v_i)$ is the number of wires connected to v_i from other RL nodes or from control ports of other BS nodes, while $C_{data}(v_i)$ is the number of wires connected to v_i from data ports of other BS nodes.

The component partitioning is divided into two steps: (1) *initial component type assignment* and (2) *type assignment for the undecided nodes*. In the first step, if bit-slice implementations are not available for the nodes, the algorithm first labels such nodes as RL. Otherwise, the nodes that meet the following two conditions are labeled as BS: (1) the component can be implemented as a bit-slice and (2) the component's bit widths are larger than a user specified threshold ($BW_{threshold}$). If these conditions are not met, the algorithm will label nodes as *undecided* type. Since the undecided nodes can be implemented as a bit-slice or a random-logic component, the algorithm takes into account the connectivity by calculating the linking cost of undecided nodes. For an undecided node v_i, if $C_{control}(v_i) > C_{data}(v_i)$ then node v_i is labeled as RL. Otherwise node v_i is labeled as BS.

The pseudo code of the algorithm is shown below. The algorithm takes an input of an RTL netlist and outputs a random-logic and a datapath components sets.

```
ALG. Component_Partitioning(G){
  for (i = 1 to n){
    if (v_i is not sliceable) then
      comp(v_i) = RL;
    else if (v_i is sliceable and
      bit_width(v_i) > BW_threshold) then
      comp(v_i) = BS;
    else
      comp(v_i) = undecide;
  }
  for (i = 1 to n){
    if (comp(v_i) == undecide) then{
      Calculate C_control(v_i) and C_data(v_i);
      if (C_control(v_i) > C_data(v_i) then
        comp(v_i) = RL;
      else
        comp(v_i) = BS;
    }
  }
}
```

Complexity analysis: Let n and m be the number of nodes and edges in the netlist. It takes $O(n)$ and $O(n+m)$ time for initial type assignment and type assignment for undecided nodes, respectively. Therefore, the complexity of component partitioning algorithm is $O(n+m)$.

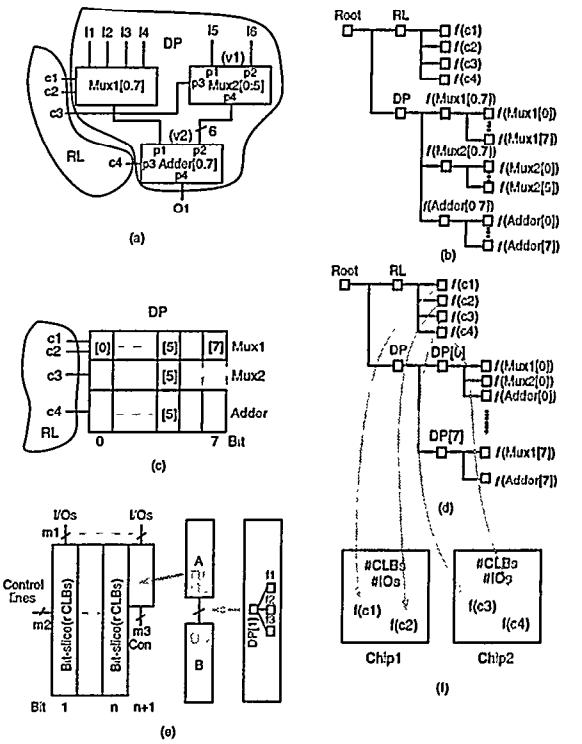

Figure 3: Functional structuring and partitioning: (a) an RT example, (b) the hierarchical function-tree, (c) the topological floorplan of the function-tree, (d) the bit-level function-tree, (e) functional packing for datapath components, (f) functional packing for random-logic components.

4.3 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. On the other hand, the logic function of a random-logic component is realized using a set of Boolean equations. A hierarchical function-tree is constructed by decomposing the functionality of the design in a top-down fashion. For example, Figure 3(b) shows the corresponding hierarchical function-tree of the design shown in Figure 3(a).

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(c). 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(d). 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(d), we can obtain its CLB netlist by invoking the logic minimizer and technology mapper [13] with the 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 [13]. 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 macro generation terminates. Otherwise, a functional partitioning procedure will be invoked which will be discussed in the next section.

Let R and DP denote the random-logic and datapath component sets, respectively. $f(R)$ and $f(DP)$ denote the logic-function sets of the random-logic and 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, R, DP){
   f(R) = Function_Generation(R);
   f(DP) = Function_Generation(DP);
   Bit_Alignment(G, DP);
   T = Bit_Level_Function_Tree(f(R),f(DP));
   CLB_IO_Estimation(T);
   if (Clb(Root(T)) ≤ CLB and IO(Root(T)) ≤ IOP) then
      Macro = Netlist(Root(T));
   else
      Functional_Partitioning(T);
}
```

Procedure *Function_Generation* generates bit-level logic functions for datapath and random-logic components. 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 *Macro*. Otherwise, the *Functional_Partitioning* procedure will be invoked to decompose the design into multiple chips.

Complexity analysis: Let BW be the average bit widths of all of the components in R and 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.

4.4 Functional partitioning

The functional partitioner partitions datapath components and random-logic components into FPGAs. The partitioner uses a bit-slice as the basic unit and packs the bit slices into FPGAs, followed by packing portions of the logic functions of one bit-slice into FPGAs. 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. For example, one bit-slice in Figure 3(e) contains r CLBs, m_1 I/O pins, and m_2 control pins. 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(e). 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 the logic functions in one bit-slice into the FPGA to improve the CLB utilization, as shown in Figure 3(e). The partitioner uses the bin-packing algorithm to pack logic functions into clusters and then performs iterative improvement using a pairwise exchange procedure. The packing procedure first partitions the datapath components into FPGAs one chip at a time, followed by packing partial logic functions of single bit-slices into FPGAs to improve their CLB utilization. After packing the datapath components into FPGAs, the unused logic and IO-pin resources of these FPGAs are then computed. Finally, the procedure packs the logic functions of the random-logic components into FPGA chips, as shown in Figure 3(f). The procedure terminates when all the components are assigned to FPGA chips.

Let T_{DP} and T_{RL} be the datapath and random-logic sub-function-trees. C denotes a set of chips used. $f(BS)$, $f(BT)$, and $f(rl)$ represent a set of logic functions of bit slices, portions of one bit-slice, and random-logic, respectively. The pseudo code of the functional partitioning algorithm is listed as follows. Procedure *Bit_Slice_Packing* determines the maximum number of bit slices which can fit into one chip. Procedure *Bit_Packing* returns portions of one bit-slice which can be packed into the chip. Procedure *Unused_Resource* computes the left-over resources of the chips. Procedure *Random_logic_Packing* packs logic functions of random-logic components into chips one at a time. If there some logic functions can not fit into any chips,

then a new chip is allocated. The algorithm terminates when all of the logic functions in the function-tree are assigned into chips.

```
ALG. Functional_Partitioning(T){
   /*Datapath Packing*/
   C = φ; i = 1;
   while (T_DP ≠ φ){
      f(BS) = Bit_Slice_Packing(T);
      f(BT) = Bit_Packing(T);
      c_i ← f(BS) ∪ f(BT);
      T_DP = T_DP - (f(BS) ∪ f(BT));
      C = C ∪ c_i; i++;
   }
   /*Random-Logic Packing*/
   Unused_Resource(C);
   while (T_RL ≠ φ){
      {f(rl),c_j} = Random_Logic_Packing(T_RL, C);
      if (f(rl) ≠ φ) then{
         c_j ← f(rl);
         Unused_Resource(c_j);
         T_RL = T_RL - f(rl);
      }
      else{
         C = C ∪ c_i; i++;
         Unused_Resource(C);
      }
   }
}
```

Complexity analysis: Let n_1 and n_2 be the number of logic-function nodes of one bit-slice and random-logic, respectively. Procedure *Bit_Slice_Packing* takes $O(1)$ time, while *Bit_Packing* takes $O(n_1)$ time. In addition, the random-logic packing procedure takes $O(n_2)$ time.

5 Experiments

We have implemented the module generator in the C programming language running on SUN and HP workstations. Currently, we have targeted two technologies: the Xilinx 3000 series and the Xilinx 4000 series chips. Using the module generator, we have generated a large set of user-defined CLB-based macro cells including multiplexers, decoders, encoders, adders/subtractors, ALUs, comparators, shifters, and logic units. Figure 4 shows the results of three ALU macros with various bit widths targeted to XC4000 chips. The first ALU contains 8 arithmetic and logic functions including addition, subtraction, increment, decrement, inversion, and, or, and 1's complement addition. The second and third ALUs contain 6 and 4 functions, respectively. The run time for generating a 32-bit ALU is less than 2 minute.

We have also tested the module generator on two benchmarking circuits and two industrial designs *Ckts 1, 2, 3, 4* which are an ALU, the elliptic filter, a datapath, and a floating-point multiplier, respectively. The bit-width of the four macros is 32 bits. We have targeted three Xilinx chips used in the emulation systems [18]: (1) XC3090 with 144 IO pins and 320 CLBs, (2) XC4010 with 160 IO pins and 400 CLBs, and (3) XC4013 with 192 IO pins and 576 CLBs. Table 1 shows the characteristics of the four macros, in which *#IOs*, *#CLBs*, *#Pins*, and *#Nets* represent the number of IOs, CLBs, equivalent gate counts, pins, and nets of the macros.

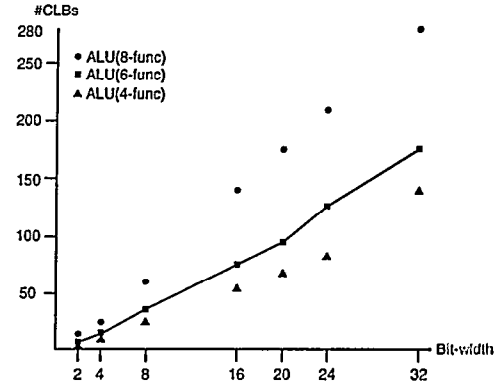

Figure 4: The results of ALU macros

Table 1: Characteristics of the benchmarking circuits.

Ckts	#IOs	#CLBs(A/B)	#Pins(A/B)	#Nets(A/B)
1	47	849/694	5321/6982	1275/1519
2	117	2223/1549	12857/14582	2739/3179
3	46	1134/861	6635/7826	1378/1798
4	109	1752/1150	9839/11590	1953/2370

A: XC3000, B: XC4000.

We have compared the partitioning results produced by our approach and a traditional approach. In the traditional approach, we first used *EmSyn* to generate a flattened CLB netlist. Then we applied the RFM algorithm to partition the flattened netlist into multiple-FPGA chips. Table 2 shows a comparison of the results with our new approach. *#P*, *IOU*, and *CLBU* represent the number of partitions, the average I/O utilization, and the average CLB utilization, respectively. The results show that our approach produced partitions with lower IO-utilization and higher CLB-utilization compared to that produced by the traditional approach.

We have also tested the two logic-emulation design approaches depicted in Figure 1 on an industrial design. The design is described in Verilog with 36 modules and 6,400 lines of Verilog code. In the first approach, we used the *HDL-ICE* [17] to synthesize the design into a gate-level netlist. Then, we used the

Table 2: Comparisons between EmGen and RFM.

Ckts	T	EmGen			RFM		
		#P	IOU	CLBU	#P	IOU	CLBU
1	A	3	.45	.88	7	.93	.38
2	A	7	.57	.99	13	.91	.53
3	A	4	.62	.89	4	.83	.89
4	A	7	.70	.78	7	.82	.78
1	B	2	.58	.87	3	.93	.58
2	B	4	.63	.98	12	.95	.32
3	B	3	.63	.72	4	.93	.54
4	B	5	.88	.58	8	.96	.36
1	C	2	.44	.60	3	.90	.40
2	C	3	.53	.89	12	.96	.22
3	C	2	.57	.75	3	.83	.50
4	C	3	.85	.66	6	.97	.33

A: XC3090, B: XC4010, C: XC4013.

Table 3: Comparisons of two logic-emulation design methods.

Method	#Gates	#Macros	#Chips	HDL-ICE	EmGen	Quest	Total	Ratio
1	27,190	0	11	00:28:50	00:00:00	00:47:19	01:16:09	1.33
2	18,615	1(1045 CLBs)	11	00:12:34	00:12:40	00:32:06	00:57:20	1.00

Quest [18] design system to perform the partitioning and system-mapping tasks. In the second approach, we manually extracted two modules from the design description. Then, we used *EmGen* to generate a macro cell of these two modules, while the rest of the design description was synthesized into a gate-level netlist using *HDL-ICE*. Finally, we used *Quest* to perform the partitioning and system-mapping tasks. Table 3 shows the comparisons of the two methods. The results show that the first approach results in total gate count of 27,190 which can be implemented with 11 XC4013 chips. Using the second approach the extracted portions of the design were implemented as a macro cell using two XC4013 chips with 1,045 CLBs. The gate count for the rest of the design is 18,615 requiring 9 chips, for a total of 11 XC4013 chips to implement the design. The results show that the run times of *HDL-ICE* and *Quest* using the second approach are 130% and 47% faster than that of using the first approach not taking into account macro generation. This gain is mainly achieved by applying synthesis and partitioning tasks to smaller-sized of designs, which drastically reduces the computation complexity of the synthesis and partitioning tasks. Macro generation in the second approach takes just over 12 minutes, thus out-performing the first approach by 33% in total run time.

6 Conclusions

In this paper, we have presented a module generator which is able to generate macro cells of arbitrarily complex functions described in HDLs. In addition, the module generator can effectively generate a multiple-FPGA macro for large macros which are not able to fit on a single FPGA chip. Using this module generator, designers are able to generate complex macros on the fly which can significantly reduce the design development time and cost.

We have conducted a series of experiments to demonstrate the effectiveness of the module generator. Furthermore, we have tested the module generation method for logic-emulation applications. We have shown that using the module generator to generate macros for portions of the design can significantly reduce the logic-emulation design processing time. The results demonstrate that the design processing time is shortened by 33% on a medium-sized industrial design when the module generation method is incorporated into the logic-emulation design flow. We believe that the run-time improvement on large designs will be to a correspondingly greater degree.

Currently, our module generator focuses on the efficiency of logic and I/O-pin utilizations of macros, which is most applicable to low-speed logic emulation applications. Further study of timing issues would be beneficial for high-speed applications. Furthermore, in order to achieve viable partitioning solutions, further study is needed of the practicality of considering routability issues during the partitioning process.

Acknowledgments

This work was supported in part by the National Science Council of R.O.C. under Grant NSC 86-2221-E-007-047 and by a grant from the Quickturn Design Systems Inc. The authors also like to thank Dr. K. C. Chu and Dr. T. C. Lin for their helpful discussions and 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.

[5] D. E. Van den Bout, "The Anyboard: Programming and Enhancements," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines 1993*, pp. 68-77, 1993.

[6] J. Gateley, "Logic Emulation Aids Design Process," *ASIC & EDA*, July, 1994.

[7] J. Kumar, N. Strader, J. Freeman, and M. Miller, "Emulation Verification of the Motorola 68060," *Proceedings of ICCD*, pp. 150-158, 1995.

[8] J. Gateley et al., "UltraSPARC-I Emulation," *Proceedings of the 32nd DAC*, pp. 13-18, 1995.

[9] G. Ganapathy, R. Narayan, G. Jorden, D. Fernandez, M. Wang, and J. Nishimura, "Hardware Emulation for Functional Verification of K5," *Proceedings of the 33rd DAC*, pp. 315-318, 1996.

[10] M. Butts, "Future Directions of Dynamically Reprogrammable Systems," *Proceedings of CICC*, 1995.

[11] *XACT libraries guide*, Xilinx, Inc., 1994.

[12] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis*, Kluwer Academic Publishers, 1992.

[13] 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.

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

[15] 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.

[16] W.-J. Fang and A. C.-H. Wu, "A Hierarchical Functional Structuring and Partitioning Algorithm for Multiple-FPGA Implementations," *ICCAD96*, pp. 638-643, 1996.

[17] *HDL – ICE User's Guide*, Version 1.0, January 1995, Quickturn Design Systems.

[18] *Quest User's Guide*, Version 4.0, January 1995, Quickturn Design Systems.

Session 4: Reconfigurable Computing

Session Chair: Jonathan Rose
Univ. of Toronto