GENERATION OF VERY LARGE CIRCUITS TO BENCHMARK THE PARTITIONING OF FPGAS

Joachim Pistorius[1] *Edmée Legai*[1] *Michel Minoux*[2]

[1] Mentor Graphics Corp. Meta Systems Division – 91975 Courtaboeuf Cedex – France
[2] Laboratoire d'Informatique – Université Pierre et Marie Curie – 75252 Paris Cedex 05 – France
Joachim_Pistorius@mentor.com Edmee_Legai@mentor.com Michel.Minoux@lip6.fr

ABSTRACT

This paper describes a new procedure for generating very large realistic benchmark circuits which are especially suited for the performance evaluation of FPGA partitioning algorithms. These benchmark circuits can be generated quickly. The generation of a netlist of 100K CLBs (500K equivalent gates), for instance, takes only two minutes on a standard UNIX workstation. The analysis of a large number of netlists from real designs lead us to identify the following five different kinds of sub-blocks: Regular combinational logic, irregular combinational logic, combinational and sequential logic, memory blocks, and interconnections. Therefore, our generator integrates a sub-generator for each of these types of netlist. The comparison of the partitioning results of industrial netlists with those obtained from generated netlists of the same size shows that the generated netlists behave similarly to the originals in terms of average filling rate and average pin utilization.

1. INTRODUCTION

The evaluation of the performance of various EDA tools is generally based on the comparison between different sets of experimental results obtained by the application of the tools on benchmark circuits. The increasing size of circuits and systems requires new partitioning algorithms capable of handling netlists with up to several million gates. Therefore, benchmark circuits of approximately the same size are needed.

Various benchmarks exist in the literature. For several reasons they do not have the characteristics required:

- Existing benchmark suites such as those collected by ACM/SIGDA, or those maintained and distributed by the CBL[1] [11] appeared in numerous papers in the past. They are, however, too small when compared with netlists to be handled by the new generation of partitioning algorithms. Indeed, the biggest netlist from the benchmark Partitioning93 contains 2904 CLBs[2] after its implementation on the XC3000 family of Xilinx FPGAs [13]. Only one benchmark from ACM/SIGDA has more than 26000 cells [17].
- The results obtained with these benchmarks for various partitioning tools are difficult to compare. As reported in [3], all of these benchmarks were originally designed for testing either placement or synthesis tools. Before using them to test partitioning tools, they have to be translated into partitioning formats. The result is that, in various publications, the same circuit has been shown with different properties.
- Real industrial netlists which have the required size cannot be used for benchmarking due to patent rights.

Recently, Alpert described in [1] the ISPD98 benchmark suite. It consists of 18 circuits varying from 13K to 210K modules in a hypergraph description format obtained by translation from internal IBM designs. Unfortunately, important information concerning circuit functionality, timing and technology has been erased by the transformation process which was obviously necessary to comply with patent rights. Due to the representation format of the circuits, the performance of the following partitioning algorithms cannot be determined correctly by using the ISPD98 benchmark suite:

- Algorithms, such as [14], which are based on the method of functional replication need information about the module functions.
- Algorithms, such as [15][16], with an objective function which aims at minimizing the length of the critical path need information about module delay and sequential modules.
- Partitioning algorithms for FPGAs, such as [6][12], are based on netlists of CLBs. These CLBs are characterized by a fixed number of input and output pins. As the benchmark circuits do not comply with this constraint, they cannot be used in the hypergraph format, but have to be transformed into netlists of CLBs. As the functionality of the various modules is not available, it is impossible to synthesize a netlist of CLBs from an ISPD98 benchmark circuit.
- A certain number of external connectors (up to 90% [1]) in the benchmark circuits are bidirectional. In order to apply a partitioning algorithm that uses the orientation of the nets, like cone algorithms [4] or algorithms based on the max-flow min-cut theorem, such as [7][20], it is necessary to transform these connectors into directional ones. The area of research is open as to how this can be done without changing the structure of the netlist.

Furthermore, due to the increasing size of netlists, which can reach several million gates nowadays, a benchmark suite of fixed size, with at most 210K cells, will rapidly become obsolete. Only a generator of heterogeneous netlists covering a large spectrum of circuit types and sizes is able to respect all required demands: It would be able to quickly create a large number of netlists of various sizes with different structural characteristics, while complying with patent rights.

[1] Collaborative Benchmarking Laboratory
[2] The CLBs (Configurable Logic Blocks) provides functional cells that implement the users's logic inside an FPGA.

The best way to evaluate the performance of various partitioning tools is to apply them to a netlist and compare the results. For FPGA partitioning, these results are often measured in terms of the number of FPGAs needed to implement a given netlist. The closer the netlist is to a real netlist, the more significant is the test. Therefore, the benchmarks have to be as realistic as possible.
In this context, a generated netlist can be said to be realistic if the application of several partitioning tools to an industrial netlist, which is supposed to have the same characteristics, leads to similar results in terms of average filling rate and average pin utilization.

Definition 1 *Average filling rate, average pin utilization:*
Let n *be the number of FPGAs needed to implement a netlist of* $|\mathrm{V}|$ *CLBs, let* k *be the number of CLBs and* c *the number of pins available in each FPGA. If* $|\mathrm{E}|$ *is the sum of external pins used on the* n *FPGAs, then* $\mathrm{F} = \frac{|V|}{(n*k)}$ *is the average filling rate or average CLB utilization of the FPGAs, and* $\mathrm{P} = \frac{|E|}{(n*c)}$ *their average pin utilization.*

New approaches to random circuit generation have been reported [5],[8],[9],[10].
Darnauer and Dai [5] presented a method for generating random circuits with a fixed number of inputs, outputs, blocks, pins per cell, and approximate Rent parameter. Their generator takes into account the constraints of digital circuits, but the consequence of the random character of the interconnection process is that the structural information of real circuits is not properly captured.
Ghosh et al. [8] extracted a wiring signature from a reference circuit in order to obtain a wiring-signature equivalent class. A wiring perturbation induces a perturbed reference circuit. The resynthesis of this perturbed reference circuit finally leads to a number of mutants which can be used as benchmark circuits. The advantage of this method is that the generated mutants are not completely random.
Hutton et al. [9],[10] developed a generator called GEN, that uses statistical information extracted from an existing design in order to generate an identical random netlist based on LUTs[3]. The suitability of netlists generated by GEN for evaluating the performance of partitioning tools has been tested with an adder and a multiplier. We partitioned a 32-bit CLA adder into units of 20 CLBs with 20 input and 20 output pins. Then, we cloned this adder with GEN and partitioned the clone into units of the same size. This procedure was repeated with a 32-bit multiplier. The constraints in this case were fixed to 60 CLBs with 60 input and 60 output pins. The results are displayed in table 1.

Circuit	#CLBs	#partitions	Filling rate	CPU time
add32	66	5	66.0%	0.1s
add32clone	62	15	20.7%	0.4s
mul32	2174	38	95.3%	18s
mul32clone	2169	76	47.6%	61s

Table 1. *Comparison of partitioning results*

They show that the generator GEN is not able to generate netlists which behave in a way similar to realistic netlists when partitioned. In fact, the number of partitions needed differs significantly from that of the original netlist while the partitioning process takes more CPU time. A possible explanation of these results is that GEN does not consider structural information which is always present in real designs (in particular the hierarchical structure).
Figure 1 shows the original netlist of the 32-bit adder which is very regular. The partitioning tool cuts horizontally with a small cutset and therefore needs a small number of partitions to contain the entire netlist.

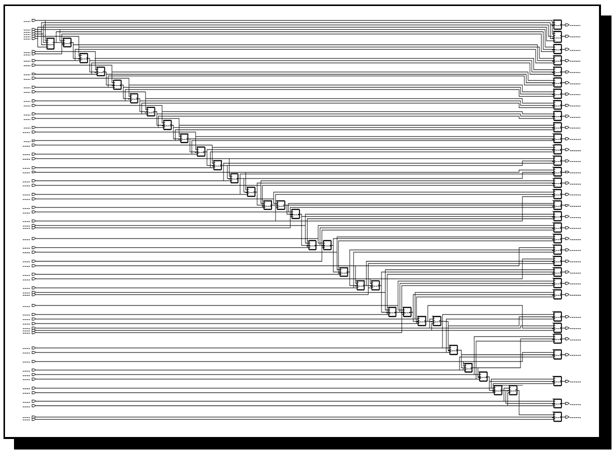

Figure 1. *Netlist of a 32-bit adder*

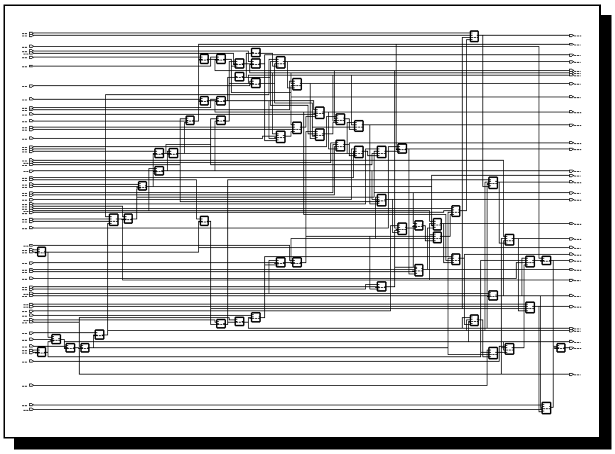

Figure 2. *Netlist of the clone of a 32-bit adder*

Figure 2 shows the clone of the 32-bit adder generated with GEN. It is rather irregular in comparison to the original netlist, and therefore difficult to partition. The number of interconnections between partitions is higher which leads to lower filling rates. These results suggest that more relevant information has to be taken into account to generate more realistic netlists, in particular structural information. Note that both drawings in figures 1 and 2 have been generated using the same graphic display tool.
We have shown that existing benchmark suites, as well as benchmarks generated by existing netlist generators, do not provide satisfactory solutions to the identified benchmarking problem in the domain of partitioning. It will be shown that our netlist generator PartGen is able to overcome this problem: It quickly generates a large number of netlists with arbitrary size. These netlists are representative of a wide variety of real designs and behave similarly to original netlists when partitioned. We can generate netlists that are easy to partition and netlists that are difficult to partition, so it is possible to examine the sensitivity, the robustness, and the performance of partitioning algorithms. Our generator also makes it possible to analyse the CPU time and the memory requirements of a given algorithm when increasing the netlist size, while leaving its characteristics unchanged. The rest of this paper is organized as follows:

[3] A LUT (look-up table) of n input pins is a memory, which can implement any boolean function of n variables.

In the next section, we identify the various types of netlists encountered in large real designs and present different generators, one for each type of netlist. The third section describes the integration of the different elementary generators into one supergenerator, which is able to generate netlists with arbitrary size and contents. In the fourth section, we present the results of our experiments with the generated netlists, followed, in the last section, by a conclusion.

2. THE PARTGEN GENERATOR

2.1. Introduction: The different types of netlists

The partitioning of a large number of netlists from real designs led us to identify five different types of designs which may be classified as follows:

- **Regular combinational logic:** Netlists of this type are very regular and generally easy to partition. Typical examples are arithmetic operators such as multipliers, or adders.
- **Irregular combinational logic:** These netlists are the "glue" in a design. They are often located around large functional blocks and are characterized by great irregularity.
- **Memory blocks:** These netlists contain memory blocks such as data cache or RAM.
- **Combinational and sequential logic:** The netlists of this type consist of both combinational and sequential logic such as cache controllers. We will therefore call them controller netlists in the following sections. They are characterized by a high average number of pins of the nets. They are often irregular and therefore difficult to partition.
- **Interconnections:** Different modules of a design communicate through netlists of this type. These interconnections are characterized by the number of nets, the number of pins on the modules and the small number of physical cells.

Every circuit is composed of one or more of these types of netlists. Because these different types of netlists feature a quite different behaviour with respect to partitioning, our generator PartGen is built by combining five sub-generators, one for each basic netlist type. In this way, the generated netlists contain instantiations of one or more netlists of one or more different types. Our approach was to develop the generators for each type mentioned above, and then to integrate them in a unique generator, PartGen. The generators and their integration into PartGen are presented below.

2.2. Regular combinational logic generator

Netlists of this type are very regularly structured. Commonly used blocks of this type are adders, counters, multipliers, or pipeline structures such as data paths.

The size of such a netlist depends on its interface. The use of an adder generator or a counter generator involves a great number of external pins of the netlist[4]. The partitioning of such a netlist leads to a relatively large number of partitions[5], which are only caused by the number of external connections, and not by the structure of the netlist. For the same reasons, the partitioning of a data path leads to a too small number of partitions. Therefore, we used a multiplier generator to create the generator for regular combinational logic.

2.3. Irregular combinational logic generator

In order to generate netlists of this type, we use the generator GEN (see [9]), mentioned in the introduction. We use the simplest possible mode for GEN, where only the number of LUT input pins and the size of the netlist to generate are given. The generated netlists contains only LUTs with 4 input pins.

[4]The 32-bit adder presented in figure 1 contains 66 CLBs and has 96 external pins.

[5]See also the average filling rates of add32 as compared with mul32 in table 1.

2.4. Memory generator

Our memory generator creates memory blocks which are 32-bits wide. In a way similar to the generation of regular combinational logic netlists, we can specify the number of memory block instances to be generated, as well as their size in terms of number of words.

2.5. Combinational and sequential logic generator

The generators mentioned above use existing generators. Our contribution is the development of a generator which creates random netlists having properties as similar as possible to real industrial controller netlists. The development is based on the analysis of an industrial netlist[6], using not only statistical, but topological information.

⋄ Generation process constraints

We analyzed a typical industrial netlist with regard to the number or cells, *flip-flops* (FF), nets, external pins, the number of input pins on the CLBs, the average fanout and the fanout distribution over the nets, as well as the length of the critical path. These characteristics are used by the generator as constraints for the generation process. The following requirements have to be respected during the generation process:

- The number of cells is the sum of the combinational CLBs and the FFs. 30 to 35% of the number of cells are configured as FFs. The rest are combinational CLBs.
- The number of primary inputs to the netlist is about 1.5% of its number of cells. The number of primary outputs is equal to the number of primary inputs[7].
- Each net is supposed to have one driver or to be connected to a primary input pin. The number of nets is therefore equal to the sum of the number of combinational CLBs, of FFs and of primary inputs.
- The number of CLB inputs follows the same distribution as the original netlist; i.e. all the combinational CLBs use input pin `I1`, nearly 99.3% use input pin `I2`, about 89.4% use input pin `I3` and approximately 66.6% use input pin `I4`.
- There is a unique clock connected to all FFs.
- The length of the critical path is fixed at 50 cells. Indeed, the analysis of other netlists with the same characteristics shows that the length of the critical path varies less with the size of the netlist.

⋄ Hierarchical decomposition constraints

The analysis of the generator GEN [9] shows that the above conditions alone are not sufficient for the correct construction of a clone which has partitioning properties similar to the industrial netlist. In fact, we need to better take into account structural information (in particular the hierarchical structure).

The hierarchy of the industrial design was analyzed by applying a bipartitioning algorithm to the original netlist [18]. The recursive application of this algorithm to the industrial netlist leads to the tree structure shown in figure 3. This hierarchical structure is reproduced during the construction process. The process starts with the generation of the macro-controllers; i.e. the macro-cells of the controller netlist, which are the leaves of the tree, and recursively climbs the branches until reaching the root. We arbitrarily respect the leaf sizes and the depth of the tree shown in figure 3, because the analyzed netlist is representative for this circuit family.

⋄ Macro-controller generation constraints

Nets with a fanout greater than three have a great probability to be cut by partitioning. On the other hand, nearly 90% of the nets

[6]The analysis of other industrial controller netlists shows that the chosen industrial controller is representative for this type of netlists. The results presented in table 4 confirm this hypothesis.

[7]This requirement helps to facilitate the generation process, described below. It does not influence the partitioning results of the generated netlist.

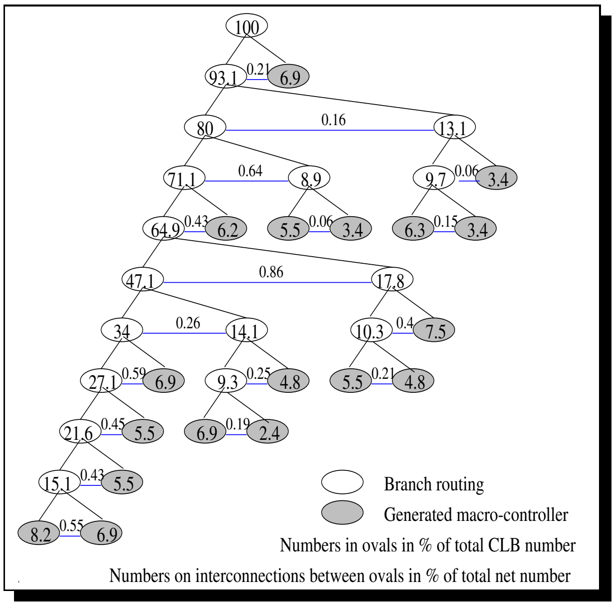

Figure 3. *Hierarchical construction process*

in the analyzed netlists have a fanout of three or less. To stay close to the observed structure of real controller designs, the interconnections are generated as follows: We start with an arbitrary linear ordering of the cells. The proximity measure for two cells v_i and v_j according to this linear ordering is $|i-j|$. The neighbor v_j of a randomly selected cell v_i is determined with a probability depending on the fanout of the net used to interconnect them. In the case of a net with fanout of three or less, we select the neighbor v_j of v_i with a probability decreasing with the distance $|i-j|$. In PartGen, we used a gaussian distribution funtion. In the case of a net with a fanout greater than three, the neighbor is determined with a uniformly distributed random function, which favours global interconnections. In addition, the following constraints must be respected while generating a macro-controller:

- Direct interconnections between primary input pins and primary output pins, as well as between FFs, are unusual and therefore forbidden.
- The number of pins of all the nets has to be greater than or equal to the sum of the primary input pins, the primary output pins, the input and output pins of FFs and the input and output pins of combinational CLBs. Indeed, in order to facilitate the generation process, we allow the fanout distribution after the interconnection process to differ slightly from the initial fanout distribution.
- Each net must have exactly one driver in order to avoid short circuits.
- Each input pin of a combinational CLB, a FF, a primary input or a primary output has to be connected to exactly one net in order to avoid non-connected pins and short circuits.
- Loops on the same cell are forbidden.
- The length of the critical path must be less than a given significant value.
- Nets with a small fanout have to be connected locally.
- Nets with a big fanout are used for global interconnections.
- Interconnections must be random.
- The construction process has to be as fast as possible.

◇ **Constructing a macro-controller**

A high level description of the construction algorithm is shown in table 2.

(1)	Generate the combinational logic cells, the sequential logic cells, the primary inputs and the primary outputs.
(2)	Generate the nets and distribute the fanout among them.
(3)	Connect all nets to a primary output pin or to one input pin of either a combinational or a sequential logic cell.
(4)	Route all interconnections of the macro-controller with respect to the constraints mentionned above.

Table 2. *Construction of a macro-controller netlist*

(1) The first step consists of the generation of all the cells of the netlist. They are concatenated in a linear chained list, one for each type of cell. A table provides quick access to a cell and facilitates the random choice of a cell during the connection process.
(2) At the second step, the total number of nets is generated. Each net randomly obtains a certain fanout according to the fanout distribution of the original real netlist. Furthermore, one clock net is generated and connected to the clock pins of the sequential cells.
(3) During the third step, each net is connected to exactly one input pin of one cell or to a primary output pin. The result of this step is a net connected to the first input pin of each cell. In this way, we avoid non-connected nets as well as non-driven cells. Note that this result is only possible because the number of primary input pins is equal to the number of primary output pins.
(4) The last step builds interconnections between the different cells.
The interconnection process is presented in detail in figure 4.

The length of the critical path is respected by the choice of a depth-first search algorithm for the cell interconnection, beginning with the primary output pins. During the routing process, we select a cell which has to be connected to another one by using a random number generator. This random number generator determines an index for the table of cells. The cell which is located at the corresponding index is chosen. The proximity of interconnections is achieved through the use of a gaussian random number generator when routing nets with a fanout smaller than three. In this case, the index of the current cell is used as an average value. Nets with a fanout greater than or equal to three are routed globally by the use of a uniformly distributed random function when selecting cells that have to be interconnected.

◇ **Interconnecting the macro-controllers**

After generating the macro-controller netlists, we have to interconnect them according to the hierarchical structure similar to the one presented in figure 3, in order to obtain one controller netlist. Experimental results show that the generation process is very reliable and extremely fast for big netlists on a standard UNIX workstation: a controller netlist of 20K CLBs takes 6 seconds to generate, while a 300K CLBs netlist (1.5M equivalent gates) takes 9 minutes.

2.6. Generator for interconnections

The aim of PartGen is the generation of one netlist containing various instances or modules, created with the generators described above. As we can now generate various netlist modules, we next need to develop an interconnection generator to connect these modules in order to obtain a single netlist.
The first step is to determine the number of primary input pins (PI_{real}) of the netlist. This number can theoretically vary between one and the sum of the primary input pins of the various modules PI_{max}:

$$1 \leq PI_{real} \leq PI_{max} \quad \text{where} \quad PI_{max} = \sum_{\forall \text{modules}} PI$$

To keep a certain depth to the netlist, we fixed the upper limit of PI_{real} as the sum of primary input pins of the modules divided by the number of modules. PI_{real} can then be determined using a

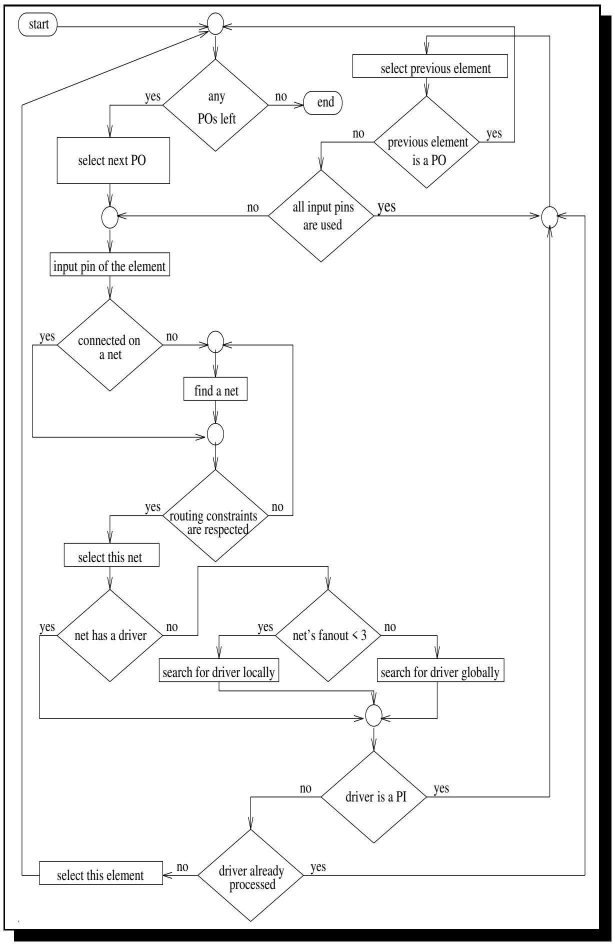

Figure 4. *Interconnection process*

uniform random number generator:

$$PI_{real} = \text{rand}[1; \frac{PI_{max}}{\text{modules}}]$$

This restricts the choice of the number of primary output pins (PO_{real}). To connect all primary output pins of the various modules, there must be a minimum for PO_{real}:

$$PO_{min} \leq PO_{real} \leq PO_{max} \quad \text{where} \quad PO_{max} = \sum_{\forall \text{modules}} PO$$

Indeed, PO_{real} has to be greater than or equal to the difference of the sum of the output pins of the modules (PO_{max}) and the maximum number of free connectors on their inputs. This number corresponds to the difference between the maximum number of primary input pins (PI_{max}) and the chosen number of primary input pins (PI_{real}); i.e. the number of input pins of the modules which are not externally driven:

$$\begin{aligned} PO_{tmp} &= PO_{max} - (PI_{max} - PI_{real}) \\ PO_{min} &= \begin{cases} PO_{tmp} & \text{if } PO_{tmp} \geq 1 \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

PO_{real} is also determined using a uniform random number generator:

$$PO_{real} = \text{rand}[PO_{min}; \frac{PO_{max}}{\text{modules}}]$$

Example 1 *Determination of PI and PO pins:*
Given three modules as presented in figure 5.
The number of primary input pins PI and output pins PO is determined as follows:

$$\begin{aligned} PI_{max} &= PI_{mod1} + PI_{mod2} + PI_{mod3} = 4 + 3 + 5 = 12 \\ PI_{real} &= rand[1; 4] = 3 \end{aligned}$$

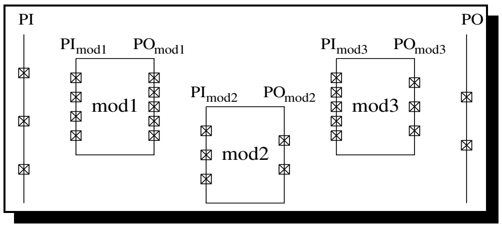

Figure 5. *Interconnection of three modules*

$$\begin{aligned} PO_{max} &= PO_{mod1} + PO_{mod2} + PO_{mod3} = 5 + 2 + 3 = 10 \\ PO_{min} &= 10 - (12 - 3) = 1 \\ PO_{real} &= rand[1; 10] = 2 \end{aligned}$$

After the determination of the netlist interface, each module must be connected to other modules and/or to primary input and output pins. The following constraints, which are the same as the routing constraints of the macrocontroller, have to be taken into account during this interconnection process:

- The number of nets is equal to the sum of the primary output pins of the modules and the primary input pins of the netlist.
- Each net must have exactly one driver to avoid short circuits or non-functional connectors.
- Each input connector of the modules and each primary output pin of the netlist must be driven.
- Loops on the same module have to be avoided.
- The process has to be as fast as possible.

2.7. Integration of the sub-generators in PartGen

The different sub-generators have been integrated in a common environment in order to facilitate their use. A netlist can be generated by specifying its size, the percentage of each netlist type that has to be generated and the number of modules of each type.

3. EXPERIMENTAL RESULTS

The validation of our generator was executed on a SUN Enterprise 5000 server with 8 Ultra Sparc II CPUs, 167MHz, and 4GB RAM. The criterion of quality for a partitioning algorithm is the average filling rate of the FPGAs needed to implement the whole netlist and the average pin utilization of the FPGAs.

We partitioned into two different FPGAs of the Xilinx 3000 family: the XC3064 with 224 CLBs and 120 I/O pins and the XC3090 with 320 CLBs and 144 I/O pins [19]. We used a partitioning tool, which recursively applies a bipartitioning algorithm to the netlist. It stops, when the size of a partition and its pin number fit the FPGA constraints. This partitioner uses a ratio-cut algorithm similar to the one presented in [18].

The generator for regular combinational circuits and the generator for memory blocks provide real, functional netlists. Therefore, they do not have to be validated. The validation of the other generators is presented in the following sections.

3.1. Irregular combinational logic generator

Irregular combinational logic is known to be difficult to partition. The validation of the use of the generator GEN for creating this type of netlist consists in showing that the output of GEN is much more difficult to partition than a regular combinational logic netlist. The experimental results are presented in table 3. The experiments show that the ratio (F_i / F_r) between the average filling rates F_i (for the irregular combinational logic netlist) and F_r (for the regular combinational logic netlist) lies in the range [0.26;0.39] for XC3064 FPGAs, and in the range [0.28;0.33] for XC3090 FPGAs. This means that the partitioning tool needs two to four times more FPGAs to implement an irregular combinational logic netlist than for a regular combinational netlist. This factor is realistic

	Irregular combinational logic				Regular combinational logic				
FPGA	#CLBs	F	P	T_cpu	#CLBs	F	P	T_cpu	F_irr / F_reg
XC3064	4417	16.2%	73.2%	12 s	5298	48.3%	64.4%	9 s	0.34
XC3090	4417	12.5%	66.5%	12 s	5298	44.7%	69.7%	8 s	0.28
XC3064	9103	17.7%	72.3%	43 s	10218	45.2%	66.4%	23 s	0.39
XC3090	9103	13.9%	67.1%	43 s	10218	42.0%	69.8%	21 s	0.33
XC3064	25595	12.7%	63.2%	281 s	24858	48.7%	71.4%	121 s	0.26
XC3090	25595	13.4%	76.5%	297 s	24858	40.7%	69.6%	109 s	0.33

Table 3. *Test of the irregular combinational logic netlist generator*

	RBA				FBB				DPRP			
Netlist	F	F/Fo	P	Tcpu	F	F/Fo	P	Tcpu	F	F/Fo	P	Tcpu
indust_1	**44.1%**	**1**	**61.7%**	**122s**	**32.5%**	**1**	**66.5%**	**12min**	**62.3%**	**1**	**89.6%**	**350s**
PartGen_best	50.5%	1.15	73.5%	111s	35.7%	1.10	65.7%	38min	52.2%	0.84	95.0%	345s
PartGen_worst	45.5%	1.03	68.3%	126s	33.9%	1.04	64.9%	30min	49.7%	0.80	96.3%	358s
GEN_best	19.3%	0.44	72.6%	177s	15.5%	0.48	62.6%	72min	28.5%	0.46	98.2%	555s
GEN_worst	18.9%	0.43	69.9%	170s	15.1%	0.46	62.5%	64min	27.3%	0.44	97.9%	561s
random_1	14.3%	0.32	74.5%	322s	11.0%	0.34	63.2%	141min	18.2%	0.29	97.9%	993s
indust_2	**52.2%**	**1**	**62.4%**	**188s**	**52.8%**	**1**	**64.8%**	**61min**	**67.5%**	**1**	**91.2%**	**562s**
PartGen_best	45.2%	0.87	71.5%	194s	33.3%	0.63	65.4%	48min	55.0%	0.81	96.7%	555s
PartGen_worst	43.4%	0.83	71.8%	218s	30.8%	0.58	62.3%	69min	52.5%	0.78	96.8%	588s
GEN_best	15.3%	0.29	63.6%	382s	14.6%	0.28	70.6%	107min	24.7%	0.37	96.9%	1018s
GEN_worst	14.7%	0.28	62.1%	388s	14.1%	0.27	68.7%	113min	24.2%	0.36	97.3%	1020s
random_2	10.4%	0.20	60.5%	615s	9.6%	0.18	62.7%	239min	16.2%	0.24	97.8%	1749s
indust_3	**42.2%**	**1**	**72.1%**	**245s**	**26.4%**	**1**	**66.9%**	**38min**	**41.0%**	**1**	**89.6%**	**1407s**
PartGen_best	44.9%	1.06	72.9%	302s	33.6%	1.27	65.0%	183min	56.5%	1.38	96.1%	1391s
PartGen_worst	42.0%	1.00	68.4%	295s	31.9%	1.21	63.7%	249min	54.8%	1.34	95.6%	1470s
GEN_best	15.2%	0.36	73.8%	539s	12.5%	0.47	63.8%	283min	21.5%	0.52	97.6%	1749s
GEN_worst	14.6%	0.35	71.1%	554s	12.2%	0.46	62.3%	284min	20.7%	0.50	98.7%	1722s
random_3	11.6%	0.27	63.3%	627s	10.6%	0.40	63.8%	454min	17.9%	0.44	97.6%	1980s

Table 4. *Validation of the generator of netlists containing combinational and sequential logic*

considering the partitioning results of these types of netlists in real industrial designs. This permits the use of GEN for generating irregular combinational logic netlists.

3.2. Combinational and sequential logic generator

To validate the generator of combinational and sequential logic, we compared the partitioning results in terms of average filling rates (F) and average pin utilization (P) of netlists generated with PartGen with the results of industrial netlists of this type. We also compared our results with those obtained for clone netlists of the industrial netlists generated using GEN[8] [9], and with the results obtained from partitioning of randomly generated netlists of the same size. The experimental results are shown in table 4. The size of the real netlists are 16893 CLBs (indust_1), 20718 CLBs (indust_2), and 22578 CLBs (indust_3). Five different netlists of the same size have been generated with PartGen and with GEN for each industrial netlist and partitioned to examine the sensitivity of both generators with respect to to the random generation process. We display only the best and the worst results obtained. Beside the recursive bipartitioning algorithm (RBA) [18], we used two other algorithms. The first one is a recursive application of the FBB algorithm described in [20] with a balance criterion of 25% of the cells in each partition. We slightly modified the algorithm by doing a breadth first search to obtain source and sink nodes as far from one another as possible. The second algorithm is the DPRP algorithm applied to an ordering obtained with the scaled cost criteria [2].

The results presented in table 4 show that the partitioning of the netlists generated with PartGen achieves an average filling rate and an average pin utilization comparable to the partitioning of the industrial netlists for three different partitioning algorithms. However, the average filling rates obtained for the randomly generated netlists, and the netlists generated with GEN, differ significantly. Indeed, the ratio (F / F_o) between the average filling rates of the generated netlists F, and of the original netlists F_o, of XC3064 FPGAs lies close to 1 for the netlists generated with PartGen. The best results were obtained for algorithm RBA where the average filling rates of the generated netlists deviates less than 17% from the original ones for all three industrial netlists. FBB and DPRP achieve good results for netlist indust_1, and satisfactory results for the two other netlists. The worst results in terms of average filling rate are 0.58 (FBB) and 1.38 (DPRP) times the average filling rate of the corresponding industrial netlist. Note that in the best case, the average filling rates obtained for the netlists generated randomly without any structural or topological information and the netlists generated with GEN do not exceed 0.44 (random) and 0.52 (GEN) times the average filling rates of the industrial netlists.

Incidentally, the ratio (T_{pg} / T_o) between the CPU times for the partitioning of the netlists generated with PartGen T_{pg} and of the original netlists T_o is close to 1 for RBA and DPRP, and exceeds 3.2 only once for FBB. However, the ratio (T_r / T_o) between the CPU time of the partitioning of the randomly generated netlists T_r and of the original netlists is under 2.5 only once.

[8] A software tool named CIRC permits the extraction of statistical information about a netlist. The generator GEN takes into account this information during the generation process. The result is a clone netlist supposed to reproduce the same statistical information than the original one.

We conclude, therefore, that the netlists generated with PartGen have nearly the same hierarchical structure as the industrial netlists. The partitioning results for the various generated netlists also prove that our generator is stable enough to reproduce nearly the same properties regarding the partitioning algorithm. This shows that our generator is able to create netlists with partitioning properties similar to industrial netlists, thus validating our approach.

3.3. Generator for interconnections

The generator of interconnections links module instances created with the sub-generators in order to obtain a unique netlist. It does not introduce physical cells into the netlist and it cannot create netlists without the other sub-generators. The interconnections of real netlists depend on the modules it contains. There are no "typical" interconnections between modules. For all these reasons, the generator of interconnections cannot be validated separately. The influence of the generated interconnections on the partitioning results remains an open area of research.

3.4. Generating benchmark circuits with PartGen

We used our generator to create 30 benchmark circuits. Their size varies from 10K to 1M CLBs. Their composition is varied. Each combination is respected; from a single type netlist to a netlist that contains at least one instance of each netlist type. The resulting netlists are flattened netlists consisting of several modules. These netlists have been partitioned using RBA. The memory blocks were treated separately by a partitioning into SRAM-banks with a capacity of 512K words $\times$ 8 bit. The rest was partitioned into XC3064 FPGAs. The partitioning results of some of these netlists are presented in table 5.

#Cells	Circuit composition	#Circuits
15000	3 mult, 1 counter	113 XC3064
20000	2 mult, 1 counter, 3 glues	226 XC3064
40000	1 mult, 1 glue, 1 memory	295 XC3064, 32 SRAM
60000	1 mult, 1 glue, 2 memories	453 XC3064, 48 SRAM

Table 5. *Partitioning mixed benchmark circuits*

4. CONCLUSION AND PERSPECTIVES

We have described PartGen, a generator of very large FPGA netlists for building partitioning benchmarks. Our approach has been validated experimentally by comparing the behaviour of the generated netlists to real netlists with respect to partitioning. For all examples treated, the results have appeared to be very similar.
Among the possible ways of improving PartGen, we mention further refinement of the netlist types by identification of sub-types and the variation of the interconnection structure to realize BUS-like interconnections between two modules. We also intend to use PartGen to create a set of benchmark circuits of large sizes which are available at *ftp://ftp-asim.lip6.fr/pub/misc/partgen*.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their very helpful comments. The first author also thanks several colleagues from *Mentor Graphics*, especially E. Hochapfel, B. Safari, C. Fouassier, G. Couaillier, G. Leclercq, C. Quennesson, P. Danois, G. Fenelon, I. Lecoeur, C. Kappler, D. Dalio, and D. Hulance, for their help in the preparation of this paper.

REFERENCES

[1] C.J. Alpert "The ISPD98 Circuit Benchmark Suite" In *Proc. ACM/SIGDA Int. Symp. on Physical Design*, pp. 85-90, 1998.

[2] C.J. Alpert, A.B. Kahng "A General Framework for Vertex Orderings with Applications to Circuit Clustering" In *IEEE Trans. on VLSI Systems 4(2)*, pp. 240-246, 1996.

[3] C.J. Alpert, A.B. Kahng "Recent Directions in Netlist Partitioning: A Survey" In *Integration: The VLSI Journal 19(1-2)*, pp. 1-81, 1995.

[4] D. Brasen, G. Saucier "FPGA Partitioning For Critical Paths" In *Proc. IEEE European Design and Test Conf.*, pp. 99-103, 1994.

[5] J. Darnauer, W. Dai "A Method for Generating Random Circuits and its Application to Routability Measurement" In *Proc.* 4^{th} *ACM/SIGDA Int. Symp. on FPGAs*, pp. 66-72, 1996.

[6] W.-J. Fang, A.C.-H. Wu "Multi-Way FPGA Partitioning by Fully Exploiting Design Hierarchy" In *Proc.* 34^{th} *ACM/IEEE Design Automation Conf.*, pp. 518-521, 1997.

[7] L.R. Ford, Jr., D.R. Fulkerson *Flows in Networks* Princeton University Press, 1962.

[8] D. Ghosh, N. Kapur, J. Harlow, F. Brglez "Synthesis of Wiring Signature-Invariant Equivalence Class Circuit Mutants and Applications to Benchmarking" In *Proc. DATE Conf.*, pp. 656-663, 1998.

[9] M.D. Hutton, J.P. Grossman, J. Rose, D. Corneil "Characterization and Parameterized Random Generation of Digital Circuits" In *Proc.* 33^{rd} *ACM/IEEE Design Automation Conf.*, pp. 94-99, 1996.

[10] M. Hutton, J. Rose, D. Corneil "Generation of Synthetic Sequential Benchmark Circuits" In *Proc.* 5^{th} *ACM/SIGDA Int. Symp. on FPGAs*, pp. , 1997.

[11] K. Kozmiński "Benchmarks of layout synthesis - evolution and current status" In *Proc.* 29^{th} *ACM/IEEE Design Automation Conf.*, pp. 265-270, 1991.

[12] H. Krupnova, A. Abbara, G. Saucier "A Hierarchy-Driven FPGA Partitioning Method" In *Proc.* 34^{th} *ACM/IEEE Design Automation Conf.*, pp. 522-525, 1997.

[13] R. Kužnar, F. Brglez, K. Kozmiński "Cost Minimization of Partitions into Multiple Devices" In *Proc.* 30^{th} *ACM/IEEE Design Automation Conf.*, pp. 315-320, 1993.

[14] R. Kužnar, F. Brglez, B. Zajc "A Unified Cost Model for Min-Cut Partitioning with Replication Applied to Optimization of Large Heterogeneous FPGA Partitions" In *Proc. ACM/IEEE European Design Automation Conf.*, pp. 271-276, 1994.

[15] L.-T. Liu, M.-T. Kuo, C.K. Cheng, T.C. Hu "Performance-Driven Partitioning Using a Replication Graph Approach" In *Proc.* 32^{nd} *ACM/IEEE Design Automation Conf.*, pp. 206-210, 1995.

[16] R. Rajaraman, D.F. Wong "Optimal Clustering for Delay Minimization" In *Proc.* 30^{th} *ACM/IEEE Design Automation Conf.*, pp. 309-314, 1993.

[17] B.M. Riess, L. Doll, F.M. Johannes "Partitioning Very Large Circuits Using Analytical Placement Techniques" In *Proc.* 31^{st} *ACM/IEEE Design Automation Conf.*, pp. 646-651, 1994.

[18] Y.C. Wei, C.K. Cheng "Ratio Cut Partitioning for Hierarchical Designs" In *IEEE Trans. on CAD 10(7)*, pp. 911-921, 1991.

[19] Xilinx, Inc. "The programmable Gate Array Data Book" Xilinx, San Jose, 1992.

[20] H.H. Yang, D.F. Wong "Efficient Network Flow Based Min-Cut Balanced Partitioning" In *IEEE Trans. on CAD 15(12)*, pp. 1533-1540, 1996.