New Synthesis Approach of hierarchical Benchmarks for Hardware Prototyping

Mariem Turki
Habib Mehrez
LIP6, Paris 6
Email: mariem.turki@lip6.fr

Zied Marrakchi
FlexRAS Technologies
Paris, france
Email: zied.marrakchi@flexras.com

Mohamed Abid
CESlab
Sfax, Tunisia
Email: mohamed.abid@ceslab.org

***Abstract*—Hardware prototyping is becoming increasingly important in the system on chip design cycle. It allows a fast hardware verification within the time to market constraints before reaching the manufacturing phase.**
However, In the prototyping cycle, designs must be synthesized and mapped into logic gates. The synthesis runtime is very big and does not satisfies the hardware prototyping goals.
In this paper we propose a new synthesis methodology which reduces the traditional synthesis runtime. This approach can automatically synthesize large and hierachical designs with a compromise between runtime and optimization. Experimentally, this new approach offers an improvement by an average of 60% compared to the synthesis runtime of an existing commercial tool.

I. Introduction

The main challenge of todays system on chip designers is to keep the cycle time as short as possible, while system complexities are increasing exponentially.
One interesting feature to decrease the time to market is to validate the increasingly large designs erlier before reaching the manufacturing phase. Currently, it is estimated that 60 to 80 percent of an ASIC design is spent performing verification [1].
FPGA-based prototyping is an important step in the creation of the final product and it is the key to the success of marketing in time. The key advantage of FPGA-based prototyping is the ability to run at high speed (sometimes at almost real-time speed) a cycle-accurate, bit-accurate model of the SoC [5]. The availability of automatic FPGA mapping tools have streamlined the design conversion process, making the path from ASIC design to FPGA implementation more straightforward.
Because the silicon area overhead of FPGA versus ASIC technology has been measured to be about 40x [6], FPGA programming technology requires that an ASIC logic design be partitioned across multiple FPGA devices to achive the necessary device logic capacity. The number of FPGAs depends on the size of the prototyped system, ranging from few [7] up to 60 FPGAs [8].
In the prototyping flow, designs have to be synthesized before being partitionned into pieces where each one can fit into a single FPGA. To synthesize a circuit, designers use the traditional flow which is a top-down based strategy since it produces better results [2]. Indeed, when starting by the highest level, the synthesizer considers the relations between the different modules in the design. Therefore, several design optimizations can be made.
On the other side, a top-down approach is not possible when synthesizing big designs because of memory and runtime limits. Synopsys [9] has added the Compile point feature which allows to reduce runtime. This synthesis feature is available with the Synplify Pro and Synplify Premier [3] products, for use with certain technology families. The compile point synthesis flow divides design into parts or points that can be processed separately. A design can have any number of compile points, and compile points can be nested inside other compile points. During synthesis, the design is first compiled, then mapped starting with the compile points at the lowest level of hierarchy in the design. After the compile points are mapped, the top-level is mapped. The fact of synthesizing the design into two iterations (bottom-up then top-down) causes additional runtime which affects the prototyping cycle.
Other methodologies are used by Design Compiler of synopsys and presented in [10]. For example, this tool uses the Time-budgeting compile methodology. Even thought it gives better runtime results, but it is hard to implement since it is very difficult to keep track of multiple scripts of each leaf. One other strategy used by Design Compiler is the Compile-Characterize-Write-Script-Recompile. This is an advanced synthesis approach, useful for medium to very large designs that

do not have good inter-block specifications defined. It requires constraints to be applied at the top level of the design, with each sub-block compiled beforehand. The subblocks are then characterized using the top-level constraints. This in effect propagates the required timing information from the top-level to the sub-blocks. The disadvantage of this approach is that the generated scripts are not easily readable. Also, lower block changes might need complete re-synthesis of entire design.
In this paper we propose a new methodology to automatically synthesis large hierarchical designs with an acceptable synthesis runtime since it is based on the bottom-up flow. The goal of this methodology is to generate a gate-level netlist with a compromise between optimisation quality and synthesis runtime. This approach is suitable to synthesize designs which are continuously modified. Indeed, when re-synthesizing a design, the designer can control the parts to be re-synthesized. And so, only the components that were modified will be re-synthesized.
The rest of the papper is organised as follows. In section 2 we give a brief description about the hardware prototyping steps and the benchmarks requirements. The proposed synthesis approach is covered in section 3. Section 4 states the caracteristiques of the used benchmarks. Finally, experiments and results are presented in section 5.

II. Hardware prototyping and benchmark requirements

Many design and verification teams are increasingly using multi-FPGA prototyping to meet ever decreasing time-to-market constraints. Fig 1 shows the hardware prototyping flow. The input, a netlist of the logic design, is transformed into a multi-FPGA configuration bitstream to be downloaded onto the prototyping board.

A. Hardware prototyping

1) Partitionning: The input design netlist is mapped to a target library of FPGA primitive. The output of the partitionning task is a synthesized netlist given as verilog files. To get those files, multiple synthesis operations are done before validating the design. As the circuits are becoming larger, the synthesis may take several minutes or even hours. Therefore it is necessary to look for other methods that accelerate the synthesis runtime.

2) Partitionning: The partitioning algorithms search for the best partition with the lowest inter- FPGA connections and highest system performance. The output of this task is a new design hierarchy which highlights the different partitions for each FPGA.

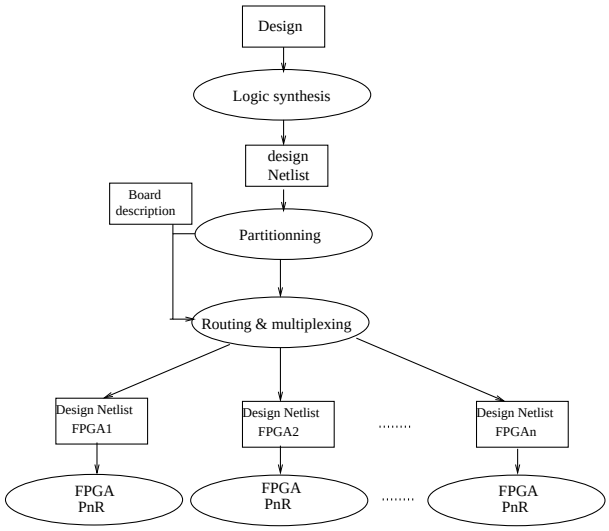

Fig. 1. Hardware prototyping flow

3) Routing and multiplexing: The routing algorithm routes inter-FPGA and I/O signals through board traces or FPGA's with the objective of minimizing signal delays. The Outputs of this task includes individual projects for each FPGA containing each the multiplexing IP for signals in the FPGA interface.

B. Benchmark requirements

To evaluate the performances of those algorithms, the used benchmarks includes specific features to allow the CAD tool developpers to test their tools.
These designs are hierarchical since the partitionner operates on high levels of hierarchy in order to reduce the partitionnig runtime and the number of treated elements. When routing the inter- FPGA signals, the routing algorithm should manage the different clocks in the design. The signals in the critical path should be given the most attention to reach the best frequency of the design under test. Therefore, most of the benchmarks contains several clocks in order to evaluate the manner that the routing algorithm deals with.
One other feature of the used designs is the heterogeneity. Indeed, if the design is symetric and contains only processors, the partitionning tool developper is not able to evaluate the performance and the intelligence degree of his tool since the partitionning is relatively obvious. For this reason, the benchmarks contains a mix of different components such as processors, coprocessors etc.

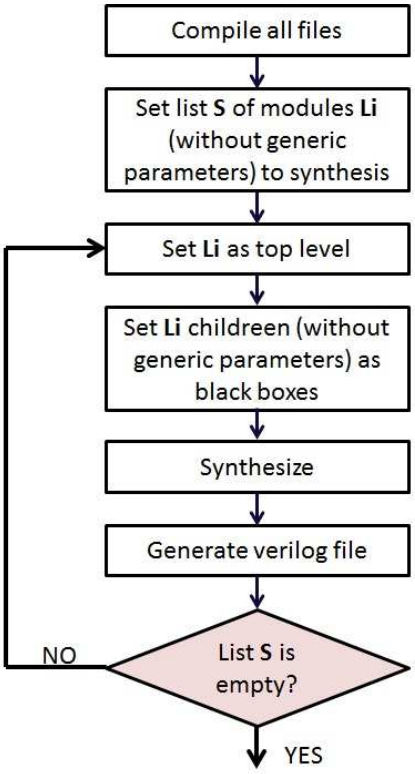

Fig. 2. Graph of the proposed synthesis approach

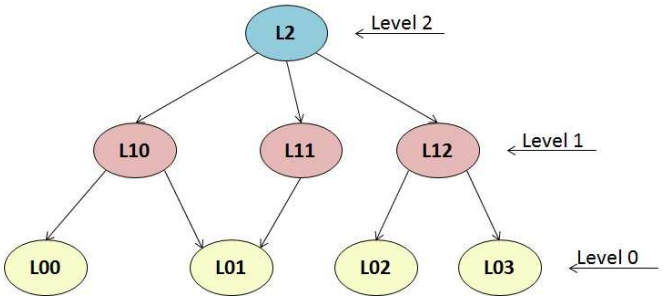

Fig. 3. Different levels of hierarchical design

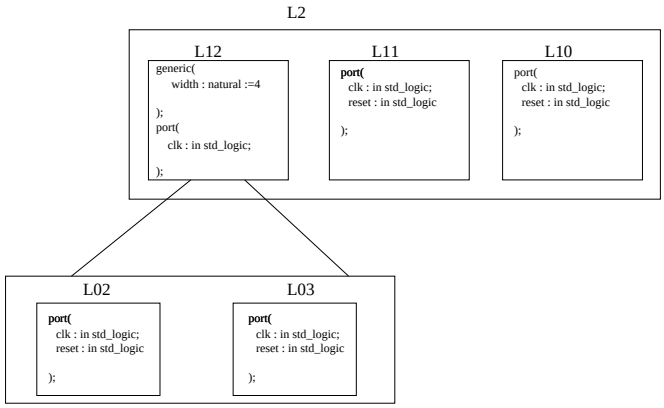

Fig. 4. Example of children list

III. The proposed synthesis approach

Most of the commercial synthesis tools use the traditional approch which is top-down based, so the tool runs the design flat. Although this approach produces the best results, designers are looking for other alternatives to synthesis their big designs because of the time and the memory limits.

Our proposed method is based on the bottom up methodology and is especially adapted for team design approach and parallel development techniques where design is split into smaller sub-projects or blocks developped independently. So, the design team can freeze portions of the design as they are completed, while continuing to work independently on the rest of the design. Each time the design is synthesized, only the components that have been modified are re-synthesized which reduce the synthesis runtime.

The proposed approach is presented in the Fig 2.

A. Logic synthesis flow

The steps detailed below are set into a script file given at the input of the synthesis tool in order to generate the netlist files of the synthesized design. Each component is synthesized independently and a netlist is generated for each component. To make this file, the designer should consider the dependancy tree of the circuit. This tree describes the hierarchy and the different levels of the design. A component is called child when it is instanciated in another component. In the example represented in Figure 3, L10, L11 and L12 are children of L2.

The first step is to compile all the files in the project. Then, the user should select the list ”S” of components ”Li” to synthesis. Two types of components should be eliminated from this list.

- The first type includes components which contain generic parameters. Indeed, each component is synthesized independently of the one in the highest level in which it is instanciated. So, at the time of synthesis, the generic parameters are not defined yet.
- The second type includes the components which are not modified since the last synthesis.

For each component ”Li” of the list S, the user defines another list ”Si” which includes all the children of ”Li” except those which contain generic parameters. Each child of ”Li” which contains generic parameters is replaced by all its children except the ones which have

generic parameters and so on. In the example in the Fig 4, the child list of the component ”L2” is defined as follows:
set L2 {L10 L11 L02 L03}.
Once all the lists are defined, the synthesis tool selects a component ”Li” from the list ”S” and ”Li” is set as a top level of the main project previously created. Then all the children of the component ”Li” which are listed in ”Si” are set as black boxes. A constraint file related to this component is added to the project. The content of this file is described in the following section.
Once all the child components are set as black boxes, ”Li” is synthesized and the output verilog netlist is generated. Finaly, ”Li” is removed from the list S. As the list S is not empty, the synthesis tool repeates the previous steps for all the remaining components until the end of all of them. At the end, the designer obtains the output netlist files of all the synthesized components. Those files will be the input of the partitionning tool.

B. Component constraint

When synthesizing a module, an sdc file which contains the timing constraints of this component should be added to the project. Design constraints specify the goals for the synthesized module. Depending on how the design is constrained the synthesis tool tries to meet the set objectives. Realistic specification is important, because unrealistic constraints might degrade the timing. Each synthesis tool has its own commands to constrain the design. Using these commands, the designers should set timing contraints which meet the component specifications. Like shown in [12], 3 types of paths must be properly set:

- Flop to flop path which specifies the clock period.
- Input to flop : Specifies the data arrival times at the specified input port.
- Flop to output : Specifies the data required times at the specified output port.

IV. BENCHMARK ARCHITECTURES

To validate the proposed methodology we used a set of hierarchical designs generated by the benchmark generator described in [11]. This generator is based on the DSX tool [16] and all the used components are part of the Soclib library [17]. In this paper, we validate the proposed approach using benchmarks which are multi-processors based architecture and contains also many coprocessors. Those architectures represent a mix of homogeneity(multiprocessors) and heterogeneity(multi-coprocessors). An example of this architecture is represented in Fig 5. This architecture contains a set of

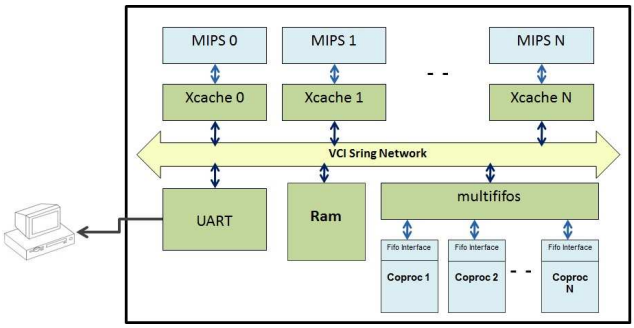

Fig. 5. Example of generated architecture

components which communicate via a VCI protocol. The example in the Fig 5 contains N processors and 3 targets: Ram, uart and a multi-fifos component. The multififos acts as a bridge between the coprocessors, each with a fifo interface, and the ring network. Large set of coprocessors can be used in order to have the biggest design. An embedded FPGA is integrated in the architecture since more recent SOC contains some field programmable cells in order to reuse a portion of the chip and to introduce new features in the design even after its fabrication. In addition, FPGA vendors and new IP developers are now offering hard embedded FPGA core that can be added into a SOC design[13], [14].
One other caracteristic of the used benchmarks is the multi-domain feature. When synthesizing the design, the user have to set constraints to each clock domain to be considered in the synthesis task. In the used benchmarks, a bi-synchronous fifo is inserted between the VCI local bus and the VCI uart component. The bisychronous fifo contains two counters to control the amount of data written and read from the fifo. Although binary counters work fine for addressing the memory, using two different clocks for the read and the write counters is problematic. A better approach for passing pointers between clock domains is to use a gray-code counter for the two fifo pointers. Gray code counters only change one bit at a time. If a synchronizing clock signal comes in the middle of a gray code counter transition, the synchronized value will either be the old or the new value because only one bit is changing at a time.
Actually, the VCI interface of the uart component is kept, and all the control signals of the VCI protocol are transmitted trough the bi-synchronous fifo.
The Fig 6 shows the connection between the network and the fifo from one side, and between the fifo and the uart from the other side.

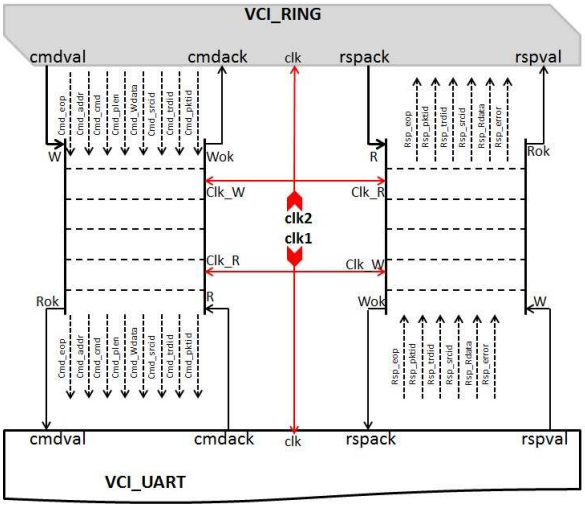

Fig. 6. Integration of an asynchronous fifo

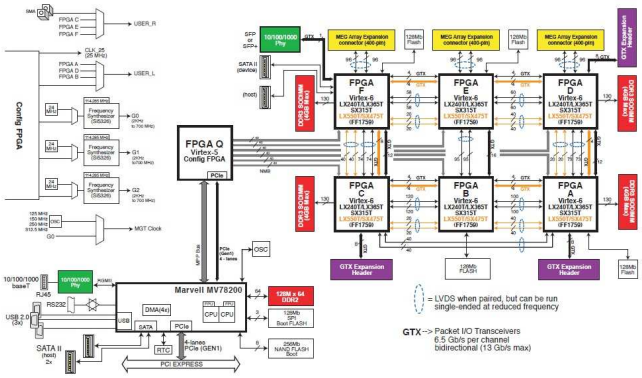

Fig. 7. Diagram of the dnv6f6pci

V. Experiments and results

We performed experiments to evaluate our proposed method comparing to the traditional flow of some commercial synthesis tools like Xilinx's XST[4] and Synopsys's Synplify Premier[3]. However, XST was not able to synthesize hudge designs which exceed some hundred of thousands of LUTs. The synthesis process always ends with an out of memory error.

The traditional synthesis flow of Synplify Premier includes two features which reduces runtime. The first one is the automatic compile point feature which divides the design into different parts or points that can be processed independently starting by the blocks at the lowest level of hierarchy. After all the compile points are synthesized, the software synthesizes the design from the top-down, using the model information for each compile point.

the second feature is the fast synthesis option which reduces significantly synthesis runtimes by a factor of 2 or 3. It accomplishes this by reducing the number of optimizations performed, so there is a trade-off in performance.

We select these two features in the Synplify Premier traditional flow, and we run the synthesis of large designs without adding our proposed script which reduce the synthesis runtime. In a second step, we run the synthesis of the same benchmarks with the Synplify Premier tool, but this time, in each design, we add our proposed script to automatically synthesis the designs with the bottom up approach.

Table I shows the details related to each benchmark such as lut size, register numbers etc..Table II show the experimental results of the tested benchmarks where runtime is measured in seconds. These results show that the synthesis runtime is improved by an average of 60% when using our proposed script. Unlike a top down synthesis, in the multiprocessor designs, the processor component is synthesized only one time. Therefore, only one output netlist is generated for the processor component. This netlist is called each time this component is instanciated in the top level file. Consequently, the runtime improvement is more significant when we increase the number of processors.

The tested benchmarks are used by Flexras technology's prototyping tools[18]. The results of implementation of these benchmarks in a multi-FPGA board are presented in table II. The board used to prototype these designs is presented in Fig 7 and contains six virtex-6 FPGA(xc6vsx475tff1759)[15]. After partitioning and routing the designs into the multi-FPGA board, information about the frequency and the number of used FPGAs are given by the prototyping tool.

VI. Conclusion

In this paper, we presented the proposed synthesis approach which reduces considerably the synthesis runtime. We used a script to automatically synthesis hierarchical designs from the lowest level of hierarchy to the highest one. This new method is especially adapted to multiprocessor designs and also during the development cycle of the circuit where modifications are possible after each synthesis. It is also adapted for the hardware prototyping where the critical time optimization is not crucial.

For the future works, we will consider the timing dependencies between the components when doing the synthesis in order to obtain the best timing results.

TABLE I
BENCHMARKS CHARACTERISTICS

Benchmark	LUTs	RAMLUTs	DSP	RAM	REG
CPU_20	143217	6192	2	21	66937
CPU_30	213524	9272	12	33	99588
CPU_50	353697	15432	25	54	164587
CPU_75	510304	20230	20	76	191200
CPU_125	879897	38532	28	130	408712

TABLE II
SYNTHESIS RUNTIME AND PROTOTYPING RESULTS OF THE TESTED BENCHMARKS

Benchmark	NB FPGA	MUX ratio	Freq(MHz)	Synplify Premier runtime	Script synthesis runtime	Imp
CPU_20	1	1	80	518s	399s	22,97%
CPU_30	3	9	27,78	864s	419s	51,5%
CPU_50	4	12	20,83	1712s	454s	73,48%
CPU_75	4	14	17,85	2233s	560s	74,92%
CPU_125	5	17	14,70	3023s	629s	79,19%

REFERENCES

[1] M. Santarini. ASIC prototyping: Make versus buy. EDN, November 21, 2005.

[2] K. Nelsen "High-Level Design Methodology Overview", Synopsys Online Documentation: Methodology Notes.

[3] Synopsys FPGA Synthesis User Guide, 2011.

[4] Xilinx. xst. www.xilinx.com/products/design tools/logic design/ synthesis/xst.htm

[5] FPGA-Based Prototyping Methodology Manual, Synopsys, 2011

[6] I. Kuon, J. Rose. Measuring the gap between FPGAs and ASICs. International Symposium on Field-Programmable Gate Array, February 2006

[7] H. Krupnova, "Mapping multi-million gate socs on fpgas: industrial methodology and experience," in Proc. of Design, Automation and Test in Europe Conference and Exhibition, vol. 2, 2004, pp. 1236-1241.

[8] S. Asaad, R. Bellofatto, B. Brezzo, C. Haymes, M. Kapur, B. Parker, T. Roewer, P. Saha, T. Takken, and J. Tierno, "A cycle-accurate, cycle reproducible multi-fpga system for accelerating mutli-core processor simulation," in Proc. of the ACM/SIGDA international symposium on Field Programmable Gate Arrays, 2012, pp. 153-162.

[9] [Online]. Available: http://www.synopsys.com/

[10] Sreesa Akella "Guidelines For Design Synthesis Using Synopsys Design Compiler" December 2000.

[11] M. Turki, Z. Marrakchi, H. Mehrez, M. Abid, "Towards Synthetic Benchmarks Generator for CAD Tool Evaluation," 8^{th} confrence on Ph.D. Research in Microelectronics and Electronics (PRIME), 2012

[12] A. Ekstrandh, W. Bell, "Evolvable Makefiles and scripts for Synthesis," SNUG(Synopsys Users Group) 1997 Proceedings, section-C1, February 1997.

[13] M. Inc, "Menta efpga core-ii data sheet brief", http://www.menta.fr/down/DatasheetBrief-eFPGA-core-II.pdf, Feb. 2009.

[14] "M2000 intros largest 90nm efpga, design and reuse", http://www.design-reuse.com/news/9614/m2000-introslargest-90nm-efpga.html, Feb. 2005.

[15] [Online]. Available: http//www.dinigroup.com/new/dnv6f6pcie.php.

[16] N. Pouillon and A. Greiner, URL=https://wwwasim.lip6.fr/trac/dsx/, 2006-2008.

[17] Soclib project: "Platform for modeling and simulation of integrated systems on chip", http://www.soclib.fr/.

[18] [Online]. Available: http//www.flexras.com