Using Hard Macros to Reduce FPGA Compilation Time

Christopher Lavin, Marc Padilla, Subhrashankha Ghosh,
Brent Nelson, Brad Hutchings, and Michael Wirthlin
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Dept. of Electrical and Computer Engineering
Brigham Young University
Provo, UT, 84602, USA
Email: {chrislavin, marcp, gho05002, brent_nelson, brad_hutchings, wirthlin}@byu.edu

***Abstract*—The FPGA compilation process (synthesis, map, placement, routing) is a time-consuming process that limits designer productivity. Compilation time can be reduced by using pre-compiled circuit blocks (hard macros). Hard macros consist of previously synthesized, mapped, placed and routed circuitry that can be relatively placed with short tool runtimes and that make it possible to reuse previous computational effort. Two experiments were performed to demonstrate feasibility that hard macros can reduce compilation time. These experiments demonstrated that an augmented Xilinx flow designed specifically to support hard macros can reduce overall compilation time by 3×. Though the process of incorporating hard macros in designs is currently manual and error-prone, it can be automated to create compilation flows with much lower compilation time.**

I. Introduction

For years, hardware designers have looked on almost despairingly at the rapid compile times of their software engineering colleagues. While their software friends perform many compile-debug-edit cycles per day, they are lucky to get one per day, or sometimes, one per week. Faster implementation times would ultimately translate into improved productivity because hardware engineers would be able to test and debug more designs per day—just like their software counterparts. Unfortunately, FPGA implementation times are not getting much faster, largely because devices keep getting bigger with every generation.

One may argue, for verification purposes at least, that compilation time can be avoided simply by using simulation to verify correct function. Indeed, where possible, simulation can and should be the tool of choice. Compile times for simulators are similar in duration to conventional software compiles and simulators provide more convenient observability than FPGA devices. However, simulation executes RTL approximately 1,000,000 times more slowly than silicon. For complex designs, e.g., software radio, radar, print rendering, etc., it often takes too long to verify functional correctness in simulation. Engineers in these situations may initially perform RTL verification with a simulator, but later, as simulation time begins to be counted in days, these engineers start loading their designs into FPGAs so their algorithms can be tested in-system, with actual data.

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 0801876.

Other researchers have tried various approaches for speeding up place and route such as router parallelization [1], multiphase placement [2], accelerated simulated annealing [3][4], clustered hierarchical placement [5] [6], and accelerated routing [7]. In contrast to the others, both [4], [5] and [6] contemplate trading faster FPGA build time for reduced quality of result (QOR). In spite of all of these efforts, however, the place and route times of commercial tools still curb the productivity of hardware engineers by limiting them to 1–2 debug cycles per day (or week) for computationally intense applications.

We believe it is unlikely that place and route times will ever be on par with software compilation times. Simply put, placing and routing circuits to meet timing and fit within the confines of an FPGA, etc., is a far more complex and demanding problem than software compilation. So what can be done? Ultimately, the goal is to reduce the time it takes to *implement* an FPGA design, i.e., go from RTL to download. One way to achieve this end is to reduce the amount of computation that is required to convert RTL to a bitstream by using pre-compiled modules.

In software, pre-compiled modules take the form of large pre-compiled libraries that need only be linked to the final executable during the compile-debug-edit cycle. Pre-compiled modules are widely used in software compilation flows and they dramatically reduce build time by *eliminating* compilation, e.g., reducing the computational effort required to build a software application.

In hardware, pre-compiled modules are typically referred to as "hard macros". A hard macro is a module that has been pre-compiled (previously synthesized/placed/routed) and stored in a library for later use by a designer. If the majority of a design consists of hard macros, the computational effort required to implement the design should be far less than if the design were compiled completely from scratch (not using pre-compiled modules). In theory at least, it should be faster to implement a design with hard macros. This paper is an early feasibility study that tests this theory and tries to find initial answers to these questions:

- Can the use of hard macros significantly reduce FPGA compilation time?
- If so, how much can the use of hard macros reduce FPGA compilation time?

The hard macro approach to design has the potential to speed up nearly all phases of the design flow, including synthesis, mapping, place, and route. It does this not by superior algorithm or parallel processing, but by reusing previous computational effort in a way similar to software but with even greater potential benefits because more complex FPGA processing flows require significantly more runtime than software compilation.

Some work has been done in effort to obtain reusability using hard macro-like cores to obtain faster FPGA compile times. Horta and Lockwood [8] demonstrated the creation of bitstream-based relocatable cores which are quite similar in nature to hard macros. Similar efforts are made in [9] where bitstream hard cores are used in a network on chip to provide accelerated logic emulation and prototyping. Unfortunately, using bitstream hard cores are much more restricted in that they must reside between configuration boundaries and require matching bus macro interfaces to be present both in the core as well as in the existing FPGA configuration. Similar work by Tessier [5] shows usage of pre-placed macroblocks accelerate place and route by 2.6× over commercial tools. However, the macroblocks did not include any routing information.

The remainder of this paper describes two experiments that were performed to test the theory that hard macros can accelerate FPGA compilation. The first experiment attempts to insert hard macros into a standard Xilinx flow. The second experiment is a "what-if" experiment that tests what might be possible if the Xilinx flow were augmented with tools specifically designed to support hard macros. The outcome of both experiments is compared to a baseline case where no hard macros are used. Results are compared for overall QOR and differences noted in size, clock rate, and compilation time.

II. Hard Macro-Based Design

In order to accelerate the Xilinx FPGA design flow, we propose using hard macros as the basic design element for all designs. That is, designs are constructed by assembling together pre-placed and pre-routed hard macro blocks.

One of the obvious benefits of using hard macros, thus, is that there is no need to run synthesis, mapping, or packing when the design is assembled[1]. The removal of these three steps of the design flow represent a signficant fraction of the overall runtime. For example, in a typical EDK MicroBlaze design, these three steps can take more than 75% of the total implementation time as seen in Table I.

An additional benefit of using hard macros as a method for faster design builds is that hard macros are relatively-placed and routed. Thus, only the hard macro needs to be placed instead of all of its individual components. This ultimately reduces the placement problem size significantly as a conventional design may have thousands of primitive instances to place, whereas a hard macro-based design may only have a few dozen hard macros to be placed. Furthermore, hard macros contain internal routing. Since the FPGA configuration routing fabric is generally homogeneous, pre-routed hard macros should be able to be placed nearly anywhere on the device. This also has the potential to significantly reduce the total number of routes to be routed in a design.

[1]It should be clear that steps equivalent to synthesis, mapping, placement, and routing must be completed to initially create the hard macros in the first place. However, in the experiments described here, our focus is to understand the benefits to be gained when hard macros are incorporated into a design.

TABLE I
Runtimes for Conventional EDK 11.4 Design flow of MicroBlaze design on Virtex 4 SX35

XST	NGDBuild	MAP	PAR	Bitgen	Total Runtime
307s (67%)	13s (3%)	38s (8%)	76s (17%)	29s (6%)	7.7 minutes

A. Experiments with the Conventional Xilinx Flow

In order to understand how well such a hard macro-based tool flow is currently supported by the Xilinx tools, we ran a series of experiments. The first task in these experiments was to build a set of hard macros which could then be assembled into a design. Xilinx documents a method for creating hard macros [10] using FPGA Editor. Although this is a manual process, FPGA Editor can be scripted to some extent [11].

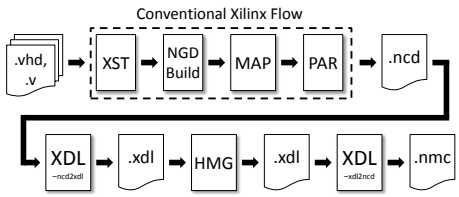

Fig. 1. Hard Macro Creation Flow

To create a set of hard macros for use in our experiments, we developed a Hard Macro Generator tool (HMG) and associated flow. Specific hard macro generation programs have been created before as in [12] and [11], however, to our knowledge, a tool to create general hard macros from arbitrary RTL has not been created before. All of our circuit manipulations are performed in XDL (XDL is a human-readable format equivalent to the more widely used NCD format). This flow leverages the first four steps of the conventional Xilinx design flow as shown in Figure 1. Using the conventional Xilinx tool flow, we create a complete, placed and routed FPGA design which contains just the circuitry for the hard macro we want to create. After turning the design into XDL, our Hard Macro Generator tool (HMG) then takes that design and converts it into a stand-alone hard macro (top level IOBs are replaced by hard macro ports among other transformations). The result is an XDL representation of the hard macro which is then converted to the Xilinx NMC format (the Xilinx hard macro counterpart format to NCD).

To create complete designs we then created structural VHDL which instanced these hard macros and then ran the

Xilinx tool flow on this VHDL from synthesis through place and route. Since the design was an interconnected set of black boxes in the VHDL there was no real synthesis performed nor was there any mapping or packing done. Rather, the bulk of the processing done by the Xilinx tools was to place and route the hard macros we had previously built. (Caveat: a final hard macro-based design flow would not use these steps of the Xilinx flow but rather would directly combine, place, and route the hard macros at the XDL level—we employed this VHDL flow for our experiments only to make it possible to quickly obtain results).

In all, a number of such tests were completed with varying results. The sample designs used in these tests included a QPSK demodulator as well as various designs created by interconnecting various open source cores found on the web. The results of these tests can be grouped into three categories:

1) For one collection of tests the design successfully placed and routed, but the resulting tool chain time was significantly longer than an equivalent process starting from a standard HDL-based design. As indicated in [10], it is known that using hard macros actually causes Xilinx PAR to run more slowly.
2) For another portion of the tests, no valid placement for the hard macros in the design could be found and the placement phase would fail. At times PAR would fail with an error message and at other times placement would simply never terminate within a reasonable amount of time.
3) Finally, for some tests, placement would complete successfully but the router would fail with an error message that it could not route all the nets in the design.

These results suggest that, in spite of the intuitive advantages that a hard macro-based design flow would seem to offer, Xilinx PAR does not work well with such a flow. The results further suggest that placement was the problem—either it failed to place the designs, or it seemed to create an unroutable placement.

B. Experiments with an Augmented Xilinx Flow

Our second set of experiments was designed to bypass the Xilinx placement step to understand the router's handling of hard macros. To do so, the original VHDL-based structural design was processed as before but only up to mapping and packing. The result was an unplaced NCD file. The hard macros were then placed by hand using a custom created XDL macro placer tool. That is, we manually chose a placement and then used an XDL-based tool of our own design to modify the un-placed XDL design file to reflect that placement. The resulting placed design was then passed to the Xilinx router (PAR) for completion.

To make an accurate measurement of runtime improvement, a baseline (non-hard macro) design was created for each test. The baseline tool runtime was determined by the total runtime of the tools from the designer's original RTL implementation (VHDL or Verilog files) to a fully placed and routed design in NCD format. The baseline measurement omits bitstream generation as this process cannot be accelerated due to the proprietary nature of the bitstream (it consumes identical run-time for both the experiment and baseline circuits). The command line parameters to the Xilinx tools were set to obtain the fastest runtime (ignoring timing constraints, standard effort levels, and avoiding timing-based MAP).

The first design is a multiplier tree made of a 20×20 bit LUT-based multiplier with a pipeline depth of 5. The Multiplier tree has 15 identical instances of the multiplier where the arrangement of the multipliers is that of a binary tree. This design was chosen to estimate the speedup that a hard macro-based flow would produce when the synthesis work load is small (the multiplier was synthesized only once).

The second design is a collection of five different cores: CORDIC, AES decryptor, Twofish (a symmetric key block cipher), FM Receiver, and Hilbert Analytic Filter. All cores were placed in the design and connected together to form a data-path with inputs and outputs connected to external IO pins. Although this design has a smaller hardware footprint than the multiplier tree, its heterogeneous nature implies that more time will be spent during synthesis and will potentially lead to increased speedup for the hard macro based flow.

C. Experimental Reductions in Run-Time

The baseline results for each design are shown in Table II. All runtimes were obtained on a desktop workstation with an Intel Core 2 Duo 3.0GHz (E6850) processor with 4GB of RAM, running Windows XP Pro SP3 and Xilinx ISE 11.4. All designs targeted a Virtex 4 SX35 (xc4vsx35ff668-10). Comparisons were made primarily on runtime, however, clock rate and hardware utilization are included as well to demonstrate the trade-offs between design runtime and QOR.

As can be seen in Table III, the hard macro designs were both built $3.1\times$ faster than their corresponding baseline build. These results offer promise to a design flow created entirely from hard macros and would enable significantly faster build times. What is also notable about these results is that they are two very different designs but both achieve the same speedup by using hard macros. This could be caused by a floor on the performance of PAR (actually, just the router since in these experiments we did our own placement). That is, both hard macro implementations took almost an identical amount of

TABLE II
BASELINE RUNTIMES FOR EACH TEST DESIGN

Design	XST	NGDBuild	MAP	PAR	Total Runtime
Mult-Tree	46.7s	9.0s	17.7s	64.2s	137.5s
Heterogeneous	80.2s	4.9s	10.4s	35.5s	131.0s

TABLE III
RUNTIMES (AND SPEEDUP OVER BASELINE) FOR EACH TEST DESIGN USING HARD MACROS

Design	Custom Placer	XDL 2NCD	PAR	Total Runtime	Speedup Over Baseline
Mult-Tree	4.3s	14.3s	25.7s	44.3s	$3.1\times$
Heterogeneous	4.3s	12.1s	25.5s	41.9s	$3.1\times$

time to run PAR. It is possible that for designs of a certain size, PAR may not be able to execute much faster than about 25 seconds. If this were the case, it may be beneficial to create a faster router for these kinds of situations.

D. Hard Macros and Quality of Results

The QOR of each circuit (as seen in Table IV) implementation is somewhat mixed. The multiplier tree baseline and hard macro implementation have surprisingly similar hardware utilization and maximum clock rates. This is an excellent result as it shows (in some cases) that using hard macros will not be much worse than using the Xilinx tools (when optimized for runtime). However, the heterogeneous design baseline produced a higher clock rate and significantly lower hardware utilization. This is probably due to the fact that the synthesizer was able to optimize some of the logic across the different block boundaries leading to lower hardware utilization and a higher clock rate. No such optimization is possible in the hard macro implementation. It must be noted, that the Xilinx tools were configured for the fastest runtime in all of the experiments and it is likely that if configured for best quality of result, clock rates of the baselines would be higher.

E. Hard Macros and Placement Time

Note that the runtimes measured in the augmented flow included the time it took to route the hard macros together, but not the amount of time it took to place them. Placement times were assumed to be negligible for these experiments based on the following analysis. Placement time for the hard macro experiments can be estimated by assuming that placement runtime behavior is approximately linear in the number of design elements to be placed. The baseline designs consisted of 4597 and 1757 slices and consumed 35.1s and 25.5s of placement time, respectively (placement time was calculated by running `par` with the `-r` option, which only runs the placer). The hard macro designs consisted of 15 and 5 blocks. Extrapolating using the linear-behavior assumption, placement times can be estimated to be 0.11s and 0.07s, respectively, for the hard macro experiments. Adding these values back into the measured runtimes does not change the result from that shown in Table III. This substantial reduction occurs because the hard macro placer only places the top-level hard macros—placement time for their constituent cells is eliminated as they were previously placed and routed. Note that the presumption of linear behavior is conservative. If the placer's behavior is super-linear, the hard macro approach will perform better than this analysis indicates, relative to the baseline.

TABLE IV
COMPARISON OF CLOCK RATE AND HARDWARE UTILIZATION OF BASELINE VS. HARD MACRO DESIGNS

Design	Slices	Clock Rate
Mult-Tree (baseline)	4592	148 MHz
Mult-Tree	4830	150 MHz
Heterogeneous (baseline)	1746	82 MHz
Heterogeneous	2741	61 MHz

III. CONCLUSION

The goal of this paper is to prove basic feasibility of using hard macros to reduce FPGA implementation time. To do so, it answered two questions: 1) Can the use of hard macros significantly reduce FPGA implementation time? (answer: yes), and 2) If so, by how much? (answer: about 3×). Two general conclusions can be drawn from the results of the two experiments. First, inserting hard cores into the standard Xilinx flow increases implementation time rather than reducing it. Though counter intuitive, this result is apparently to be expected, based upon information available from the Xilinx knowledge base [10]. Second, significant reductions in FPGA implementation time are possible. The second experiment indicates that it may be possible to reduce implementation time by up to 3.1×. A 3× reduction in FPGA implementation time means that an engineer can complete 6 debug cycles per day instead of two, for example. Further, this second experiment demonstrated it should be possible to augment the Xilinx flow with a hard-macro-aware placer to achieve these reductions in implementation time. It is not necessary to develop a completely new tool chain.

REFERENCES

[1] P. K. Chan and M. D. F. Schlag, "New Parallelization And Convergence Results For NC: A Negotiation-Based FPGA Router," in *FPGA '00: Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2000, pp. 165–174.

[2] Y. Xu and M. Khalid, "QPF: Efficient Quadratic Placement For FPGAs," in *Proceedings of the IEEE International Conference on Field-Programmable Logic and Applications. IEEE, Los Alamitos, CA*, 2005.

[3] V. Betz and J. Rose, "VPR: A New Packing, Placement And Routing Tool For FPGA Research," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*. Springer-Verlag London, UK, 1997, pp. 213–222.

[4] C. Mulpuri and S. Hauck, "Runtime And Quality Tradeoffs In FPGA Placement And Routing," in *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*. ACM New York, NY, USA, 2001, pp. 29–36.

[5] R. Tessier, "Fast Placement Approaches for FPGAs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 7, no. 2, pp. 284–305, 2002.

[6] Y. Sankar and J. Rose, "Trading Quality For Compile Time: Ultra-Fast Placement For FPGAs," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. ACM New York, NY, USA, 1999, pp. 157–166.

[7] J. S. Swartz, V. Betz, and J. Rose, "A Fast Routability-Driven Router For FPGAs," in *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 1998, pp. 140–149.

[8] E. L. Horta and J. W. Lockwood, "Automated Method to Generate Bitstream Intellectual Property Cores for Virtex FPGAs," in *Proc. Field Programmable Logic.2004*, 2004.

[9] Y. E. Krasteva, F. Criado, E. d. l. Torre, and T. Riesgo, "A Fast Emulation-Based NoC Prototyping Framework," in *RECONFIG '08: Proceedings of the 2008 International Conference on Reconfigurable Computing and FPGAs*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 211–216.

[10] "AR #10901 - 6.1i FPGA Editor - How do I create a hard macro?" http://www.xilinx.com/support/answers/10901.htm.

[11] "Using Three-State Enable Registers in 4000XLA/XV, and Spartan-XL FPGAs (XAPP123 v2.0)," Xilinx Inc., Tech. Rep., January 2002.

[12] C. Claus, B. Zhang, M. Huebner, C. Schmutzler, J. Becker, and W. Stechele, "An XDL-based Busmacro Generator for Customizable Communication Interfaces for Dynamically and Partially Reconfigurable Systems," in *Workshop on Reconfigurable Computing Education at ISVLSI 2007*, Porto Alegre, Brazil, May 2007.