XDL-Based Module Generators for Rapid FPGA Design Implementation

Subhrashankha Ghosh, Brent Nelson
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Dept. of Electrical and Computer Engineering
Brigham Young University
Provo, UT, 84602, USA
Email: brent_nelson@byu.edu

***Abstract*—XDLCoreGen is described, a module generator framework which directly generates placed and routed hard macros in XDL. XDLCoreGen is intended to be used in a rapid prototyping flow such as HMFlow, which achieves short FPGA implementation times by bypassing the conventional Xilinx tool flow and directly assembling designs from pre-built hard macros. The structure of XDLCoreGen is described and its unique cache-based router is highlighted as a key component to achieving extremely fast module generation times. Testing results are provided to demonstrate XDLCoreGen's ability to generate fully placed and routed hard macros in milliseconds.**

I. Introduction

Over the years researchers have proposed two broad categories of methods for reducing FPGA design time. The first focuses on *front-end design entry* — using high-level synthesis (HLS) tools and/or predefined circuit building block libraries to accelerate design entry and verification. One way they do so is by hiding many low-level FPGA details from the designer. However, advances in this area will never significantly reduce design time unless we also make progress in a second area: that of reducing *design implementation time* (synthesis, place, and route). In fact, reducing implementation time may be even more important than reducing front end design time since the entire design debug process usually consists of many repeated debug-modify-recompile iterations. Thus, reductions in implementation time will be multiplied many times over during the course of a design.

One method for accelerating design implementation is to use hard macros (preplaced and routed circuit building blocks) and quickly assemble them into finished circuit layouts, bypassing the conventional synthesis, place, and route tool flow. One tool based on this approach is HMFlow [1], [2] which assembles finished FPGA designs from pre-compiled hard macros. However, HMFlow relies on the conventional Xilinx tool flow — a very time consuming process — to initially create its hard macro building blocks. To fully harness the power of hard macro implementation approaches, however, will require a method for more rapidly creating hard macros. In response, this paper introduces XDLCoreGen, an XDL-based module generator tool which directly generates placed and routed XDL hard macros. As will be shown, XDLCoreGen can generate placed and routed hard macros building blocks in a few milliseconds, enabling rapid prototyping approaches based on hard macros to implement FPGA designs in seconds or minutes. In the balance of the paper we describe XDLCoreGen and its structure and demonstrate its use in conjunction with the HMFlow rapid prototyping flow.

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

II. Background

The HMFlow environment, shown in Figure 1, processes designs entered using System Generator [3]. HMFlow parses the design's .mdl file and consults its hard macro cache ("HM Cache") to determine which building blocks it already has available and which must be generated. If any must be generated, it creates them using its hard macro generation tool ("Generic HMG" in the figure), which uses the Xilinx implementation flow with specially-created constraints to synthesize, place, and route each building block into a specified rectangular area of the FPGA fabric. HMFlow then converts that into a relocatable XDL hard macro. Once all the needed blocks have been created, HMFlow places and routes them into a final XDL design. A challenge with this flow is that creating a required hard macro may take many minutes of CPU time. XDLCoreGen is designed to reduce this hard macro generation time to milliseconds.

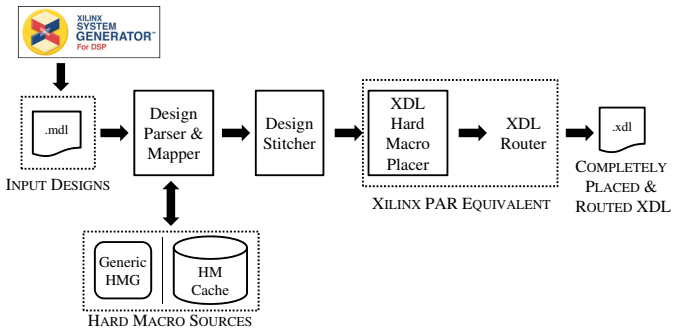

Fig. 1: The HMFlow Process

Others have recognized the value of XDL and hard macros

for circuit generation. References [4], [5], [6] describe generating circuits in XDL for producing specialized bus and communications structures for partial reconfiguration architectures. Additionally, Torc [7] is a tool which leverages XDL and provides an open-source general design flow for Xilinx FPGAs. Our work, while similar to these, is ultimately focused on the creation of an extremely fast FPGA design prototyping flow to reduce design development times. That said, there are many similarities among these projects in that all use XDL for design manipulation and have a notion of hard macros for reducing design complexity.

A. The Xilinx Design Language (XDL)

XDLCoreGen generates its hard macros in the Xilinx Design Language (XDL), which is a textual representation of a Xilinx design and may represent it at any stage of implementaton: mapped, placed, and/or routed. XDL explicitly represents the Tiles, Wires, and PIPs (programmable interconnection points) in a design and provides an alternative to Xilinx's native netlist format (NCD) which is proprietary. An XDL example is shown in Figure 2.

```
design "adder4bit" xc4vsx35ff668-10 v3.2 ,                                  DESIGN
cfg "
_DESIGN_PROP::PK_NGMTIMESTAMP:1294258207
_DESIGN_PROP::PIN_INFO:Gateway_In(3):/adder4bit/PACKED/adder4bit/Gateway_In(3)/Gateway_In(3)/
PAD:INPUT:3:Gateway_In(3\:0)
_DESIGN_PROP::PIN_INFO:Gateway_In(2):/adder4bit/PACKED/adder4bit/Gateway_In(2)/Gateway_In(2)/
PAD:INPUT:2:Gateway_In(3\:0)
_DESIGN_PROP::PIN_INFO:Gateway_In(1):/adder4bit/PACKED/adder4bit/Gateway_In(1)/Gateway_In(1)/
PAD:INPUT:1:Gateway_In(3\:0) ... "

# ============================================================               MODULE
# The syntax for modules is:
#    module <name> <inst_name> ;
#    port <name> <inst_name> <inst_pin> ;
#    instance ... ;
#    net ... ;
#    endmodule <name> ;
# ============================================================
# ============================================================
# MODULE of "adder_5_bits_56"
# ============================================================
module "adder_5_bits_56" "adder_3" , cfg "";
port "GLOBAL_LOGIC0_inport_BX" "adder_1" "BX";
port "addsub1i_0_inport" "adder_1" "F3";
port "addsub1i_1_inport" "adder_1" "G3";                                  INSTANCE
...
inst "adder_1" "SLICEL",placed CLB_X1Y37 SLICE_X1Y74 ,
cfg " BXINV::BX BYINV::#OFF CEINV::#OFF CLKINV::#OFF COUTUSED::0 CY0F::F3
CY0G::G3 CYINIT::BX DXMUX::#OFF DYMUX::#OFF F:LutEquation_3:#LUT:D=(A4@A3)
F5USED::#OFF FFX::#OFF FFX_INIT_ATTR::#OFF FFX_SR_ATTR::#OFF FFY::#OFF
FFY_INIT_ATTR::#OFF FFY_SR_ATTR::#OFF FXMUX::FXOR FXUSED::#OFF
G:LutEquation_2:#LUT:D=(A4@A3) GYMUX::GXOR REVUSED::#OFF SRINV::#OFF
SYNC_ATTR::#OFF XBUSED::#OFF XMUXUSED::0 XUSED::#OFF YBUSED::#OFF
YMUXUSED::0 YUSED::#OFF CYMUXF:CarryChain_CYMUXF_2:
CYMUXG:CarryChain_CYMUXG_3: XORF:CarryChain_XORF_2:
XORG:CarryChain_XORG_3: "
;
...
net "carryChain_net_3" ,                                                    NET
  inpin "adder_3" CIN ,
outpin "adder_2" COUT ,
pip CLB_X1Y37 COUT3 -> COUT_N3 ,
;
...
endmodule "adder_5_bits_56" ;
```

Fig. 2: An Example XDL File

For purposes of this discussion, the key parts of the XDL file in the figure are its Instance and Net declarations. The *Instance* statement contains multiple attribute strings which configure it. Each attribute has the format:

attributeName:logicalName:value

The attribute name specifies the part of the primitive being configured; the value specifies a configuration value. The instance in Figure 2 contains this attribute string:

F:LutEquation_3:#LUT:D=(A4@A3)

which describes the equation implemented by the slice's F LUT.

An XDL *Net* declaration represents a connection between FPGA primitives. It always contains pins to represent the net's source and sink connections and specify its logical structure. It may also, optionally, contain PIPs which represent how it is physically routed through the FPGA fabric. As shown in Figure 2, the single Net in that design contains a PIP connecting a wire from the COUT pin on instance "adder_2" to the CIN pin on instance "adder_3" (neither of these two instances are shown in the figure due to space restrictions).

Finally, the XDL *Module* construct provides hierarchy in a design by encapsulating a portion of that design. When XDLCoreGen creates a hard macro, it does so as an XDL Module.

XDLCoreGen is built on top of RapidSmith, a JAVA based API previously developed to enable easy manipulation of XDL files. More information can be found in [8], [9].

III. XDLCoreGen Overview

XDLCoreGen is a Java library of module generators which creates macros using a two step process: (1) creating and configuring FPGA primitives and (2) placing and routing them together into a finished hard macro. The class hierarchy for XDLCoreGen is shown in Figure 3. Starting from the left, XDLCoreGen provides classes to create and configure individual FPGA primitives. The *Logic* class uses these classes to create configured primitives needed by module generators. The *HardMacro* class provides methods for placing and interconnecting these building blocks and the *Router* class routes them. In the middle of the figure, the module generator classes use the functionality of all these other classes to assemble placed and routed hard macros into final designs.

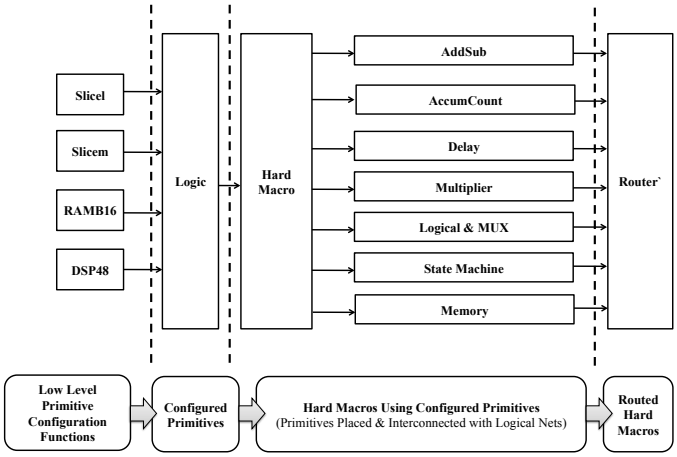

Fig. 3: XDLCoreGen JAVA package class hierarchy

A. Primitive Configuration

A primitive is configured by modifying its attribute strings. Examples of attributes from the slice of Figure 2 include

"F:LutEquation_3:#LUT:D=(A4@A3)" (specifies the F LUT equation) and "CYINIT::BX" (selects the source of the carry in signal to be from the "BX" input pin). XDLCoreGen includes a collection of such classes to help in the creation and configuration of primitives. Examples include the *SLICEL*, *SLICEM*, *RAMB16* and *DSP48* classes in Figure 3.

B. The Logic Class

Although the primitive configuration classes provide the ability to create and configure FPGA primitives, using them to create full hard macros is tedious. To address this, the *Logic* class shields the module generator classes from the details of primitive creation and configuration. When requested, *Logic* creates useful pre-configured primitives required by the module generator classes. Whenever *Logic* creates a configured primitive it also caches a copy of it so that it can be quickly cloned and returned when another copy is needed.

1) An Example — Configured Slices For Building Adders: As an example, several configured SLICEL primitives are needed to build a multi-bit adder macro. The adder starts with a slice with its carry in coming from the BX pin, its carry out turned on, and its LUTs programmed to do the add function. The middle portion of the adder is made from multiple slices with their carry in/carry out wires connected to neighboring slices. Finally, the adder ends with a slice which terminates the carry chain. An example of a middle slice, showing how the various resources are configured (in bold), is shown in Figure 4. These and other building blocks are created by the *Logic* class for building subtractors, adder/subtractors, multiplexors, array multipliers, shift registers (delay lines), memories, and basic logic gates.

C. The HardMacro *Class*

The *HardMacro* class provides methods to place and logically interconnect configured primitives. Its placement methods accept parameters so that the resulting hard macro can be constrained into a specified rectangular region if desired. Since carry chains go upward and shift chains go downward in Xilinx FPGAs, the placer also accepts a parameter to control the direction of placement (bottom-to-top or top-to-bottom). Any time the next block would be placed outside the bounding box's Y boundaries, the macro is folded and continued in the next column to ensure that the desired module shape is produced. Figure 5 shows the placement of 11 SLICEMs (which have downward pointing shift chains) into a bounding box. If this were a placement of 11 SLICELs instead, they would be ordered bottom-to-top and would use all four slice locations in each CLB since SLICELs can use either SLICEM or SLICEL sites.

Once the instances in a hard macro have been placed they must be interconnected logically to specify to the downstream router how they are to be routed. The *HardMacro* class provides methods for doing this.

D. The Module Generator Classes

The actual module generator classes extend the *HardMacro* class and use its and the *Logic* class's methods to construct

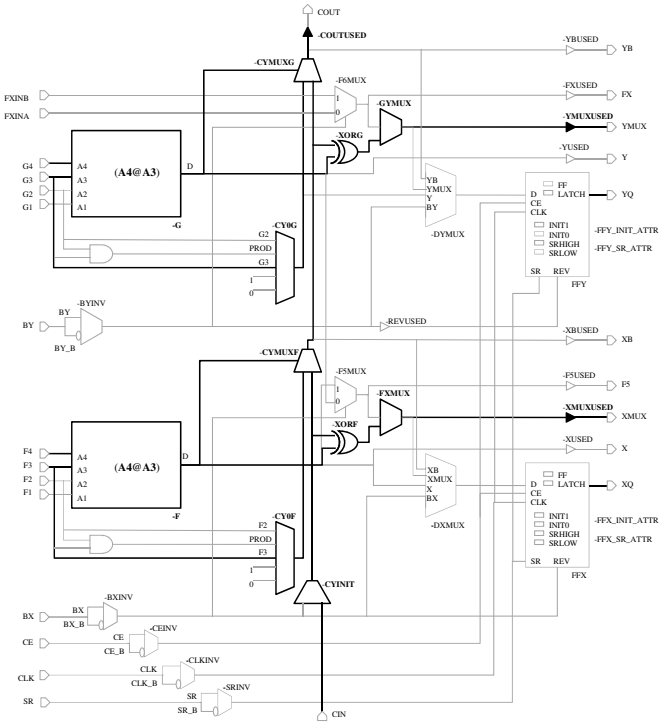

Fig. 4: SLICEL Configured As An Adder Mid Slice

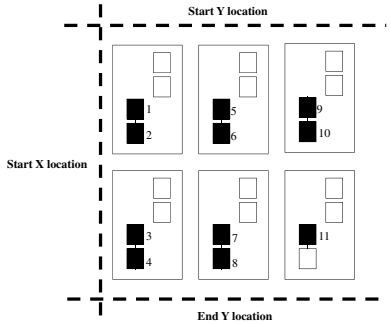

Fig. 5: Placement of 11 SLICEM primitives

and assemble configured primitives into a placed hard macro. The current list of module generator classes provided in XDLCoreGen includes:

- *AddSub*: creates adders and subtractors
- *AccumCount*: creates accumulators and counters
- *Multiplier*: creates both logic- and DSP-based multipliers
- *Delay*: creates shift registers (delay lines)
- *Logical*: creates wide logic gates and multiplexors
- *FSM*: creates state machines
- BlockRAM: creates memories

The hard macro generators accept an extensive set of parameters, increasing their usefulness in a variety of designs. These parameters allow for the specification of input and output precision (bitwidths), radix point placement, shape

(bounding box), latency (registered or not), etc. The FSM module generator takes as input the format accepted by JHDL [10], a tabular PLA-like description format.

IV. Cache-Based Routing in XDLCoreGen

The final step in creating a hard macro is to physically route its internal nets. Since the majority of XDLCoreGen's hard macros have a regular, replicated structure, we decided (for speed reasons) to create our own template-based router rather than rely on either Xilinx's router or HMFlow's maze router. The approach chosen was to pre-generate a collection of routes needed by XDLCoreGen's hard macros and store them in a cache for later use. Additionally, it was observed that the majority of the routing fabric within Xilinx parts is regular. This allowed for the use of *relocatable routes* within the cache — routes which have relative endpoints and which can be used anywhere within the FPGA fabric as needed. For example, all of the vertical length-two connections using a specific column wire in the entire fabric can be represented using a single route entry in the cache.

Figure 6 shows an example route in a Xilinx FPGA. With only a few exceptions (such as carry chains), all interconnections between primitives go through a succession of switchboxes. Within each switchbox, PIPs make connections between the wires entering/exiting the switchbox. In the figure, the illustrated net starts at the YB pin in the left CLB tile, passes through two switchboxes, and terminates at the BX input on a SLICEM in the right CLB tile in the figure. This same routing path is replicated everywhere across the chip and so only needs to be represented once in the cache.

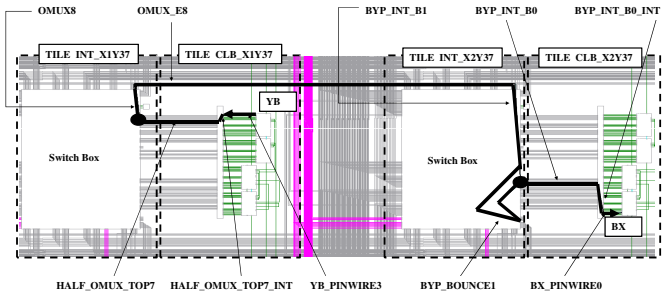

Fig. 6: A Route in an FPGA

For hard macros with regular layouts, a list of the required routes was created. The maze router of [8] was then used to pre-create these routes and they were then placed into a routing cache. To save space, the routes were pruned by removing their end pins so they represent only switchbox to switchbox routes. The advantage of this is that even though the endpoint pin names are different for different primitives, all the switchboxes are the same. Thus, many fewer routes are needed in the cache. After a route is selected for use, the cache router finishes it by simply adding the wires needed to go from the outpin to the switch box on one end, and from the switch box to the inpin on the other. The result was a drastic reduction in cache size.

For state machines, the needed routes cannot be predetermined since FSM routing is irregular. Thus, the approach taken was to create a different routing cache for FSMs which contains all the routes from a given point that reach a certain number of switchboxes in all directions. This is shown for a distance of two in Figure 7.

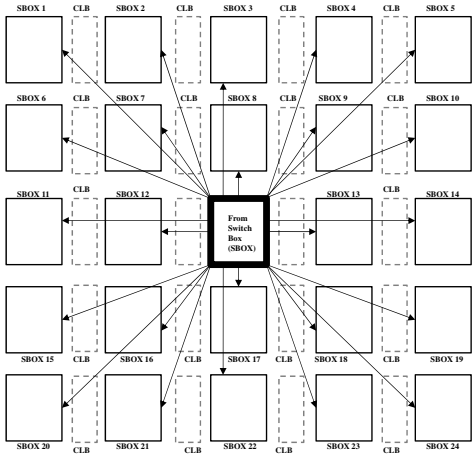

Fig. 7: A Distance 2 Cache Of Routes

As shown in Table I, three different route caches were eventually created. For all cached routes, multiple alternate paths were created and placed in the cache to provide additional options for the cache router. During execution, XDLCoreGen loads only the route cache needed by the called module generator.

A representation of a typical route is shown in Figure 8. Routes are indexed in the cache using the names of their switchbox endpoints and the XY offset they traverse. When a route is selected from the cache for use it is offset by the coordinates of the location it is to be used at within the FPGA fabric. Route selection during execution is done by XDLCoreGen in a greedy fashion — the shortest route between two points is selected by the router from the cache. However, there may be cases where no route will be found. This can happen if a hard macro is very large or if the candidate routes have already been used. In these cases, the router will simply leave that route unconnected, to be connected later by the final design router when it routes the hard macros together. This is a nice side benefit of the XDL approach we have taken — the final design router, be it the Xilinx router or another router, will complete any unrouted hard macro internal nets as a part of its operation. However, this has very rarely been required.

TABLE I: The Route Caches

Cache	Size	# Routes	Load Time (ms))
CacheOfRoutesSimplest	2KB	32	0.3
CacheOfRoutesMultiplier	46KB	540	8.1
CacheOfRoutesFSM	299KB	12,804	59.3

Wire From Name: HALF_OMUX_TOP1 Wire To Name: IMUX_B1 X*(tiles)*: +2 Y*(tiles)*: 0		Start Wire (row, col), End Wire (row, col)
	Option #1	BOUCE1 *(33,15)*, IMUX_B1 *(33,15)* E2BEG4 *(33,15)*, BOUNCE1 *(33,15)* OMUX_E8 *(33,15)*, E2BEG4 *(33,15)* HALF_OMUX_TOP1 *(33,13)*, OMUX8 *(33,13)*
	Option #2	BYP_BOUCE7 *(33,15)*, IMUX_B1 *(33,15)* BYP_INT_B7 *(33,15)*, BYP_BOUNCE7 *(33,15)* OMUX_E13 *(33,15)*, BYP_INT_B7 *(33,15)* HALF_OMUX_TOP1 *(33,13)*, OMUX13 *(33,13)*
	Option #3	BYP_BOUCE2 *(33,15)*, IMUX_B1 *(33,15)* BYP_INT_B2 *(33,15)*, BYP_BOUNCE7 *(33,15)* BYP_BOUNCE1 *(33,15)*, BYP_INT_B2 *(33,15)* BYP_INT_B1 *(33,15)*, BYP_BOUNCE1 *(33,15)* E2MID6 *(33,15)*, BYP_INT_B1 *(33,15)* OMUX9 *(33,13)*, E2BEG6 *(33,13)* HALF_OMUX_TOP1 *(33,13)*, OMUX9 *(33,13)*

Fig. 8: A Typical Cache Entry

V. Results

Several tests were conducted with XDLCoreGen targeting a Xilinx Virtex-4 SX35 FPGA. The tests were run on a Windows XP system with an Intel Core 2 Duo CPU running at 2.66 GHz with 3.5 GB RAM. In all of our testing XDLCoreGen was used in conjunction with HMFlow to produce complete placed and routed designs. However, we emphasize that XDLCoreGen is not limited to use with HMFlow — it provides a Java API for the creation of hard macros and can return completed macros in the form of either a Java data structure or an XDL file.

The first set of tests compared the time required by XDLCoreGen to create a variety of hard macros against the current HMFlow method. Since HMFlow currently uses the Xilinx tools to synthesize/place/route hard macros, these tests provide a speed comparison between CoreGen's and XDLCoreGen's hard macro generator processes.

These tests were done by creating a System Generator design and then running HMFlow twice on the design. In the first run HMFlow used System Generator and CoreGen to synthesize, place, and route each required hard macro and then it converted each to XDL. In the second run HMFlow instead simply used the XDLCoreGen API to directly create each required hard macro as needed. The results are shown in Table II where it can be seen that the speedup for XDLCoreGen was between about $1,000$ and $5,000$.

TABLE II: Hard Macro Generation Time Comparison

Type(bitwidth)	System Generator (s)	XDLCoreGen (s)
AddSub(4)	78.200	0.016
AddSub(16)	78.253	0.017
Accumulator(4))	78.770	0.020
Accumulator(16)	78.781	0.021
Counter(4)	78.508	0.020
Counter(16)	78.533	0.022
Multiplier(4)	79.047	0.026
Multiplier(16)	79.015	0.072
Delay(8×64)	78.423	0.016
FSM (3 states)	76.547	0.073

Figure 9 shows where the time is spent by the accumulator module generator. In this case, the single most time consuming step was the creation and configuring of the primitives for performing the accumulator logic. Somewhat surprisingly, placement and routing of the hard macro was a small fraction of the total time required, as was the time required to load the routing cache. In these tests the cache router was able to internally route all the hard macros shown in the table, even the state machine which contained many Nets with multiple sink pins ("rats nest" routes).

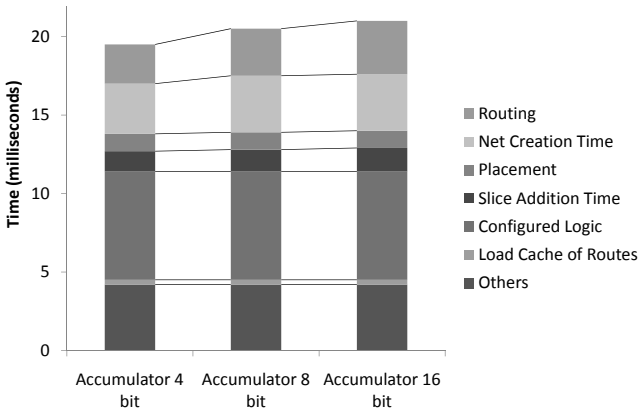

Fig. 9: Accumulator Hard Macro Generator Time Distribution

In a second set of tests we measured the performance of just the router portion of the XDLCoreGen tool for internally routing hard macros. Its speed is compared to that of the HMFlow maze router in Table III. These results suggest that the template routing approach used is generally superior (for hard macros)) to that of even a highly speed-optimized maze router such as is found in HMFlow. However, for the largest and most irregular hard macros it is inferior to HMFlow's router. For comparison, the speed of the Xilinx router was also measured and, in each case, took about $3000ms$ to route each hard macro's internal nets, suggesting startup time dominates its run time for small circuits like hard macros.

TABLE III: Routing Speed Comparison (all times are in ms)

Type(width)	HMFlow Router	Cache Router
Addsub(16)	18.1	2.3
Accum(16)	18.2	3.7
Counter(16)	18.1	4.0
Multiplier(16)	48.0	51.4
5-in NAND(16)	18.4	3.6
Delay(8) (depth=64)	18.1	2.3
FSM (3 states)	19.3	62.3

A. Application-Based Testing

This section will demonstrate the capabilities of XDLCoreGen by showing its use in the development of a 1K point FFT design. This design was originally created using System Generator, and thus relied on Xilinx's CoreGen for the creation of its building blocks (adders, multipliers, memories, etc). Since the module generators in XDLCoreGen support the CoreGen parameters used on the building blocks in this design, it was thus a simple matter to use XDLCoreGen to generate the macros in the FFT design instead of having System Generator create them. The FFT design contained a total of 51 unique

hard macros, each of which was used multiple times for a total of 335 hard macro instances in the design.

The experiment was based on four different processing scenarios: (1) the conventional System Generator flow was used, (2) HMFlow was used starting with its cache empty (meaning it would need to generate all needed hard macros using System Generator before it could place and route them), (3) HMFlow was used starting with its cache full (meaning it already had all the hard macros needed in its cache), and (4) HMFlow was used starting with its cache empty but called on XDLCoreGen to generate the needed hard macros.

The results of these experiments are summarized in Table IV. The processing time of scenario 2 (HMFlow w/empty cache) was dominated by the time for the Xilinx tools to synthesize, place, and route each hard macro separately and for HMFlow to convert each one to XDL. The processing time of scenario 3 (HMFlow w/full cache) represents the best case situation for HMFlow — creating a design wholly from pre-existing hard macros. Scenario 4 is similar to Scenario 2 except that XDLCoreGen was used to create the needed hard macros rather than the Xilinx tools. The processing time for scenario 4, while not as short as scenario 3 was still much shorter than either of the first two scenarios, illustrating the value of XDLCoreGen as a viable module generator method for a rapid prototyping environment. The implementation time difference between scenarios 3 and 4 represents the time required to create the 51 unique hard macros needed by the design.

TABLE IV: FFT Design Experimental Results

Scenario #	Impl. Time (sec)	$T_{clk}(ns)$
1 - Conventional flow	120.0	7.9
2 - HMFlow w/empty cache	3711.3	14.0
3 - HMFlow w/full cache	7.7	14.0
4 - HMFlow w/XDLCoreGen	8.2	12.1

As previously documented [2], the performance of hard macro based circuits is less than those produced by the conventional Xilinx flow, a tradeoff made to achieve short runtimes. However, compare the clock periods for scenarios 2, 3, and 4. Here, the only difference is that the Xilinx tools created the macros in scenarios 2 and 3 while XDLCoreGen created the macros in scenario 4. In either case, the same tool (HMFlow) placed and routed them into a final design. The resulting clock period was shorter for the XDLCoreGen designs, which is counter-intuitive based on the simple algorithms it employs for creating macro layouts.

An examination of the layouts created by XDLCoreGen shows it generates hard macro layouts very similar to those produced by the Xilinx tools in most cases. For adders, counters, and accumulators, it stacks slices in columns and connects them with fast carry logic. A closer comparison, however, shows why scenario 4 produced faster circuits. In scenarios 2 and 3, HMFlow uses blocks created by System Generator, which it then converts to hard macros. To influence the block's internal placement and therefore the resulting hard macro shape, HMFlow generates an area constraint for the Xilinx tools to use. This area constraint generation process is unaware of the natural organization that other modules might possess (columnar organization with LSB at the bottom and MSB at the top, for example). Indeed, this was the case with this FFT design — the internal cells of the MUX blocks were placed in a somewhat arbitrary fashion within the bounding box by the Xilinx tools and thus resulted in longer routes to neighboring arithmetic blocks. To test the hypothesis that this was the cause of the clock rate difference, scenario 4 was rerun using scenario 2 and 3 MUX macros and scenario 4 hard macros for everything else. The resulting clock rate was $13.8ns$, suggesting the reason for the slower clock rate in scenarios 2 and 3 was due simply to the MUX block layout.

VI. Conclusions and Future Work

XDLCoreGen, an XDL module generator for rapid implementation of hard macros has been described. The results given show that it can generate fully placed and routed hard macros in milliseconds compared to the minutes required by alternative approaches. While the use of XDLCoreGen was demonstrated in conjunction with the HMFlow tool, it could be used independently of that tool to very rapidly produce XDL-based hard macros for other tool flows.

While we believe these first experiments demonstrate its value, additional tasks can help increase its usefulness. The first task would be to increase the number of module generators it contains and increase the parameterizations available for each. A second task would be to investigate how to structure it to simplify retargeting it to new FPGA families. Finally, we believe its cache based routing approach can be adapted to work in other CAD tool contexts.

References

[1] C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin, "Using Hard Macros to Reduce FPGA Compilation Time," in *Proceedings of the 20th International Workshop on Field-Programmable Logic and Applications (FPL'10)*, August 2010.

[2] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping," in *Field-Programmable Custom Computing Machines (FCCM), 2011 19th IEEE Annual International Symposium on*, May 2011, accepted, to appear.

[3] Xilinx, "System Generator Getting Started Guide," http://www.xilinx.com/support/sw_manuals/sysgen_gs.pdf, March 2008.

[4] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder A Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAS," in *Proceedings of the 18th International Workshop on Field-Programmable Logic and Applications (FPL'08)*, September 2008.

[5] 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," *Workshop on Reconfigurable Computing Education at ISVLSI*, 2007.

[6] A. Oetken, S. Wildermann, J. Teich, and D. Koch, "A Bus-Based SoC Architecture for Flexible Module Placement on Reconfigurable FPGAs," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, september 2010, pp. 234 –239.

[7] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: Towards an Open-Source Tool Flow," in *Proceedings of the 19th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011.

[8] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, B. Hutchings, and M. Wirthlin, "RapidSmith: A Library for Low-level Manipulation of Partially Placed-and-Routed FPGA Designs," Brigham Young University, http://rapidsmith.sourceforge.net, Tech. Rep., 2010-2011.

[9] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapid Prototyping Tools for FPGA Designs: RapidSmith," in *Field-Programmable Technology (FPT'10). International Conference on*, December 2010.

[10] BYU Reconfigurable Computing Laboratory, "State Machine Generators," http://www.jhdl.org/documentation/users_manual/fsm.html, 2003.

[11] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, B. Hutchings, and M. Wirthlin, "A Library for Low-level Manipulation of Paritally Placed and Routed Designs," http://rapidsmith.svn.sourceforge.net/viewvc/rapidsmith/ trunk/-doc/TechReportAndDocumentation.pdf, 2010.