StML: Bridging the Gap between FPGA Design and HDL Circuit Description

Dustin Peterson, Oliver Bringmann
Embedded Systems
Department of Computer Science, Faculty of Science
Eberhard Karls Universitaet Tuebingen
Tuebingen, Germany
{dustin.peterson, oliver.bringmann}@uni-tuebingen.de

Thomas Schweizer, Wolfgang Rosenstiel
Computer Engineering
Department of Computer Science, Faculty of Science
Eberhard Karls Universitaet Tuebingen
Tuebingen, Germany
{thomas.schweizer, wolfgang.rosenstiel}@uni-tuebingen.de

***Abstract*—FPGA circuit implementation is a unidirectional and time-consuming process. Existing approaches like the incremental synthesis try to shorten it, but still need to execute the whole flow for a changed circuit partition. Other approaches circumvent process stages by providing bidirectional mappings between their results. In this paper we propose an approach to provide a bidirectional link between an FPGA design and its HDL code. This link enables the circumvention of the most time-consuming stages (synthesis, mapping, placing, routing) of the FPGA circuit implementation. We implemented our approach in a Java-based EDA tool library, called *Static Mapping Library (StML)*. We demonstrate its applicability by means of hardware debugging and an RTL-based injection of permanent faults, built on top of the StML. Experimental results illustrate that a mapping coverage between 98.5% – 100.0% can be obtained, which substantiates the feasibility of this approach. Further experiments illustrate a controllable tradeoff between area overhead, circuit granularity and mapping granularity. With the finest mapping granularity, the area overhead has been between 1.8% and 60.2% for RTL-based circuits. The speedup of the proposed fault injection method has been estimated to be up to 6x for the tested circuits.**

I. Introduction

Nowadays, the complexity of circuits is growing steadily. While smaller structure sizes in CMOS technology allow the integration of an increasing number of components into ICs, FPGAs benefit from growing array sizes and deeply integrated components on-chip. In turn, the growing complexity leads to an increasing effort that needs to be handled by the EDA (Electronic Design Automation) tools.

In terms of FPGA circuit development, the increasing effort, handled by the EDA tools, is spread throughout the whole FPGA design flow (synthesis, mapping, placing, routing and bitstream generation), which is bringing the high-level circuit description to a low-level implementation in the behavioural, the structural and the physical domain, as illustrated in the Gajski-Kuhn chart [10]. Altogether, they may consume tens of minutes up to several hours, thus playing a central role regarding the development costs, especially for verification and debugging because of their repeated invocation of the already mentioned processes. In order to decrease these development costs, there is the necessity to accelerate or circumvent parts of the design flow.

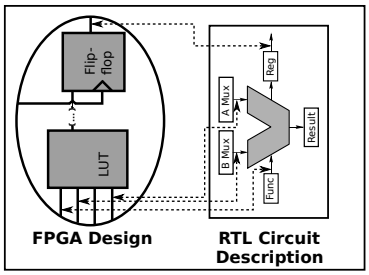

Fig. 1: Static mapping: Bidirectional link between the high-level HDL source and the low-level FPGA design

In this paper, we present an approach that enables the circumvention of the synthesis, mapping, placing and routing by constructing a bidirectional link between an FPGA design and its HDL sources. The bidirectional link, that we call *static mapping*, maps elements from the HDL circuit description directly to primitives from the FPGA design, as illustrated in Figure 1. Therefore, it provides detailed knowledge about the fully placed and routed FPGA design with respect to the circuit description. Furthermore, it enables the circumvention of the design flow from synthesis to the routing process for small selective changes, due to a direct transfer of locations from the circuit description to the design implementation.

This paper is structured as follows: In this section we briefly describe the related work. In Section II we introduce the preliminaries of our method. Section III extensively describes the details of our approach. In Section IV two use cases, taking advantage of the presented approach, are illustrated. Section V outlines and discusses the experimental results. Finally we conclude our work in Section VI.

A. Related Work

There are several approaches, proposing the acceleration of parts of the design flow. Brand et al. [5] introduced the *incremental synthesis*, that is reusing previously gained synthesis results in further synthesis runs. Soni et al. [15] proposed a methodology to accelerate the generation of a bitstream. Guccione et al. [11] introduced JBits – a tool for independently adjusting an existing bitstream for Virtex-2 devices. These are illustrated in Figure 2 (a).

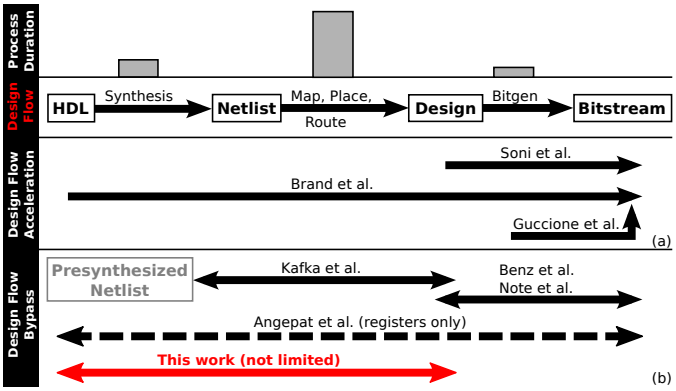

Fig. 2: Related work: (a) Methods that are partially accelerating the FPGA design flow (b) Methods that are mapping or bypassing design flow stages

Although each approach reduces the time consumption partially, they are inefficient when applying selective changes to the HDL code. Brand et al. have to execute the whole design flow for the specific partition, Soni et al. are restricted to adjust low-level design changes to the bitstream only and JBits is only reasonable for HDL-independent changes.

Further approaches provide bidirectional links between different stages of the FPGA design flow as illustrated in Figure 2 (b). Kafka et al. [12] presented an approach to preserve the circuit structure through the synthesis, thus enabling a mapping between a design and an intermediately presynthesized netlist. Benz et al. [4] introduced an approach to regenerate an FPGA design from a bitstream using a mapping between both. While they are regenerating FPGA designs from bitstreams, Note et al. [14] proposed a methodology to construct a mapping between a bitstream and its design. Angepat et al. [1] illustrated a method to create a link from a circuit description at RTL (Register-Transfer Level) to the design and the bitstream, using a vendor tool-generated resource allocation file. Due to the content of this file, the approach is limited to registers-only (so, no combinatorical logic may be mapped). In this work, we propose a new approach to close the gap between a circuit description and its FPGA design that extends the approach of Angepat et al. without its limitations.

B. Main Contributions

We present a method to construct a bidirectional link between a placed and routed FPGA design and its RTL-based circuit description. The method has been implemented in our Java-based *Static Mapping Library (StML)*. We demonstrate the applicability examplarily regarding two use cases: A method to inject permanent faults at RTL into the FPGA design and an approach for a debugging environment. We described and did the integration of the fault injection method into the StML.

II. Preliminaries

In this section, we introduce the basics of the FPGA design flow. We further present the libraries, used in our implementation. Finally, the sample circuit, which is used to demonstrate our approach, is introduced.

A. FPGA Design Flow

FPGAs are fine grained configurable architectures that consist of an array of configurable logic elements. In the Xilinx terminology, we are referring to, the logic elements in the array are called *instances*. An instance is either highly configurable, such as a *slice*, or less configurable, like a *digital signal processor*. In turn, it may consist of fine-grained configurable primitives such as *look-up tables* (LUT), *flip-flops* (FF), *latches*, or *multiplexers* (MUX). Instances are interconnected using the global configurable switch matrix.

In our research, we focused on the Xilinx Virtex-4 FPGA-family and on a workflow, based on *Synopsys Synplify G-2012.09* and *Xilinx ISE 14.1i*. The workflow starts with an FPGA-synthesizable circuit description in a HDL like Verilog or VHDL. Synplify synthesizes the design into a gate-level netlist, using an FPGA-specific technology library based on FPGA primitives. Subsequently, the synthesized netlist is passed to the Xilinx toolchain. First, it is *mapped* to a specific FPGA device and allocates resources in the two-dimensional array of logic elements. Afterwards, the design is *placed and routed* what is done by *par*. It places the mapped resources to specific instances and configures the switch matrix to interconnect them correctly. Finally, a low-level bitstream of the design is created, which is uploaded to the FPGA then.

B. Resulting Files

The flow results in different files that we use to generate the static mapping. Those are as follows:

- .NCD Native Circuit Description: FPGA design in a closed unprocessable format.
- .XDL Xilinx Design Language: FPGA design in an open plain-text format [3].
- .BIT Bitstream: SRAM Configuration which is uploaded to the FPGA.
- .VM Verilog Simulation Netlist: Is a Verilog-based synthesized netlist, especially designed for simulation purposes. Its modules, that are partially representing FPGA primitives, are implemented by the Xilinx UNISIM-Library.

In the upcoming sections we will especially use the placed and routed FPGA design (XDL) and the simulation netlist (VM), in order to create the sought static mapping between the circuit description and the FPGA design. The bitstream generation requests an NCD-based FPGA design – the Xilinx-tool *xdl* is therefore used to convert between XDL and NCD.

C. Java Libraries in Use

In order to analyze the input files, we utilized two Java libraries: *Rapidsmith*[1] [13] and *EDAUtils VerilogParser*[2].

Rapidsmith is a Java-based library that supports the development of CAD-tools for FPGA designs. It provides an

[1] http://rapidsmith.sourceforge.net/
[2] http://www.edautils.com

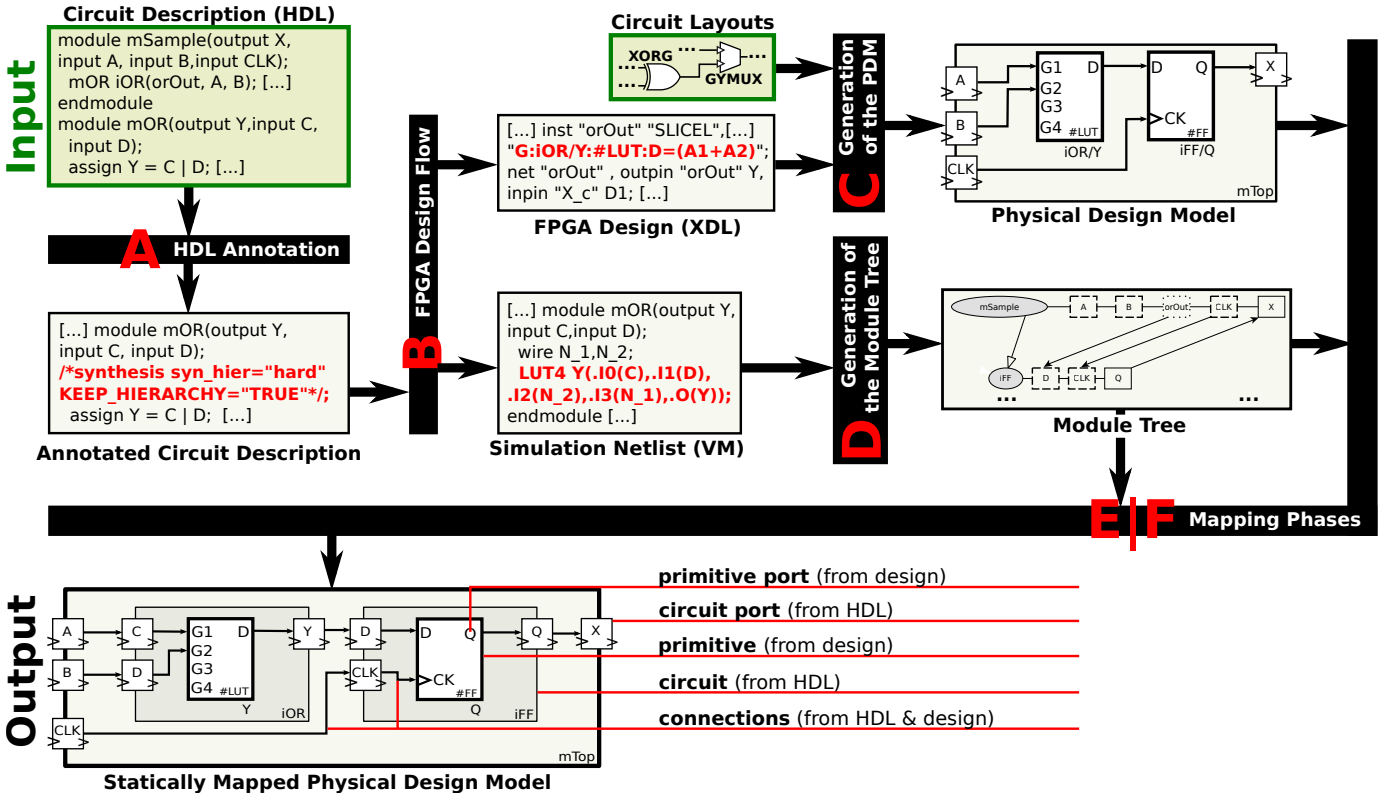

Fig. 3: Flow of our mapping procedure: Each illustrated process is described in the corresponding subsections from A to F

implementation for parsing and editing XDL and has rudimentary bitstream support. The library supports most of the Xilinx FPGA families such as Virtex-4, Virtex-7 or Spartan-6. The library is used for parsing, storing and editing the XDL.

EDAUtils provides a parser for Verilog. It performs a direct conversion of the plain text Verilog file into a class model where each language construct is represented by one class (e.g. modules, module instances, assignments, sensitivity lists). The library takes over the parsing of the simulation netlist.

D. Sample Circuit

For a vivid explanation of our approach, we visualize it by means of the following circuit description:

```
module mTop(output X,input A,input B,input CLK);
  wire orOut;
  mOR iOR(orOut, A, B);
  mFF iFF(X,orOut,CLK);
endmodule
module mOR(output Y,input C,input D);
  assign Y = C | D;
endmodule
module mFF(output Q,input D,input CLK);
  reg Q;
  always @(posedge CLK) begin Q <= D; end
endmodule
```

III. Static Mapping between Circuit Description and FPGA Design

Imagine, a developer implements a circuit and wants to verify the whole circuit or components with respect to fault tolerance. Therefore, several faults have to be integrated into the circuit description and its behaviour during execution should be monitored. Generally, there are two main options: Software-based simulation and FPGA-based emulation.

While simulation suffers from its low performance, it enables the tester to get full control to the simulation details, e.g. the monitored signals. Additionally, small changes of the design may be passed fast and easy to the simulation environment.

In contrast, emulation performs much better, but the tester does not get full control of the emulation internals. The FPGA acts as a black box and hides implementation details, thus making it difficult to monitor RTL-based signals. Unfortunately, small changes of the circuit description have to pass the whole time-consuming FPGA design flow down to the design implementation and bitstream generation.

In order to solve these issues, we introduce an approach to construct a bidirectional link between the FPGA design and its circuit description. It enables the circumvention of the synthesis, mapping, placing and routing and regenerates implementation internals. Therefore, we combine modules and ports from the circuit description with a layout representation of the design, that we call physical design model. It leads to a direct link between fully placed & routed design and the circuit description, as illustrated in Figure 1. We call this link *static mapping*.

Figure 3 depicts the the *mapping procedure*, that generates the static mapping. In the following sections, we describe the participated processes A – F.

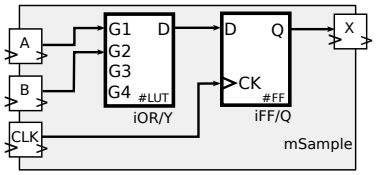

Fig. 4: Illustration of the Physical Design Model (Sample Circuit)

A. Annotation of the Circuit Description

Out of the box, FPGA design (XDL) and simulation netlist (VM) do not correspond in terms of structure. In order to enable their structural equivalence, the compiler needs to preserve the hierarchy. *Synopsys Synplify G-2012.09* as well as by *Xilinx ISE 14.1i* provide options to preserve the hierarchy throughout the whole FPGA design flow either due to annotation of the circuit description or by additional compiler commands. This causes the design and the simulation netlist to correspond in their structure.

B. FPGA Design Flow

The circuit description passes the design flow as described in section II-A. Due to preserving the structure, the simulation netlist is able to bridge the gap between the design and the circuit description. It contains the hierarchy in terms of modules, instances and ports from the circuit description as well as primitives and wires from the design.

The code listed below represents a simplified version of the sample circuit's Verilog-based simulation netlist:

```
module mTop(output X,input A,input B,input CLK);
  wire orOut; mFF iFF(.Q(X),.D(orOut),CLK(CLK));
  mOR iOR(.Y(orOut),.C(A),.D(B));
endmodule
module mOR(output Y,input C,input D);
  wire N_1,N_2;
  LUT4 Y(.I0(C),.I1(D),.I2(N_2),.I3(N_1),.O(Y));
endmodule
module mFF(output Q,input D,input CLK);
  FD Q(.Q(Q),.D(D),.C(CLK));
endmodule
```

The XDL-based design of the sample circuit has been adjusted in the same way:

```
design "mTop" xc4vlx100ff1513-10 v3.1;
inst "X_c" "OLOGIC",placed IOIS_LC_L_X0Y167
  OLOGIC_X0Y334, cfg " CLK1INV::C D1INV::D1
  INIT_OQ::0 OFF1:iFF/Q:\#FF";
inst "orOut" "SLICEL",placed CLB_X1Y167
  SLICE_X0Y335, cfg "G:iOR/Y:\#LUT:D=(A1+A2)
  YUSED::0";
net "A", cfg " _BELSIG:PAD,PAD,A:A";
net "A_c" , outpin "A" I, inpin "orOut" G1;
net "B", cfg " _BELSIG:PAD,PAD,B:B";
net "B_c", outpin "B" I, inpin "orOut" G2;
net "CLK" , cfg " _BELSIG:PAD,PAD,CLK:CLK";
net "X" , cfg " _BELSIG:PAD,PAD,X:X";
net "X_c", outpin "X_c" OQ, inpin "X" O;
net "orOut", outpin "orOut" Y,inpin "X_c" D1;
```

C. Generation of the Physical Design Model

The FPGA design (XDL) is a plain-text configuration that provides two access paths: *Instance/Attribute* and *Net/Pin*. Because signals from the circuit description (HDL) cannot be assigned to attributes of the XDL configuration, it is indispensable to parse and convert the design to a physical layout that we call *physical design model (PDM)*. The PDM is continuous physical layout representation of the FPGA design with named and fully interconnected primitives. It is created by merging a specific XDL with layouts of the corresponding FPGA family (here: Virtex-4), which is done automatically by the StML. Therefore, the FPGA layout data of the most important instances, that are documented in the Virtex-4 user guide[3], have been gathered: SLICEL, SLICEM, ILOGIC, OLOGIC and TIEOFF. The layout data describe the placement of primitives and ports as well as the connections in between them. We extracted the existing primitives: look-up tables, flip flops, multiplexers, boolean gates, supplies (fixed-1), grounds (fixed-0), RAM-modules, ROM-modules, blocks for controlling read and write access to the RAM-modules (WSGEN) and undocumented primitives (blackboxes). These primitives including their port specifications are created and used in the PDM.

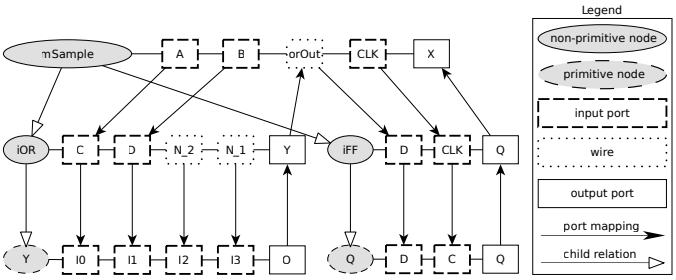

Fig. 5: Illustration of the module tree for the simulation netlist (sample circuit)

In order to build the PDM, the design is parsed by *Rapidsmith*. Rapidsmith provides a class model that maps directly to the XDL structure: `Instance`, `Attribute`, `Net`, `Pin`, `PIP`. The generated model is merged with the FPGA layout data by creating and connecting the configured primitives in the PDM. Each of its primitives gets a *primitive name* from the XDL, containing the full path of the primitive inside the module tree. Furthermore, each primitive contains input and output ports, called *primitive ports* (e.g. input port *D* in a flip-flop). Primitives need to be interconnected by unidirectional *connections* from a source primitive port to a sink primitive port. Connections from primitives within one instance are created using the FPGA layout data merged with the instance configuration. In contrast, the connections between primitives from different instances is originated using the global switch matrix, which is implemented by *nets*. Figure 4 illustrates the PDM of the sample circuit.

D. Generation of the Module Tree

Simultaneously, we parse and convert the simulation netlist into a tree structure that we call *module tree*. It contains hierarchy information of modules, instances, ports, wires and port mappings. In order to build up the module tree, the Verilog parser introduced in section II-C, has been extended. The

[3]http://www.xilinx.com/support/documentation /user_guides/ug070.pdf

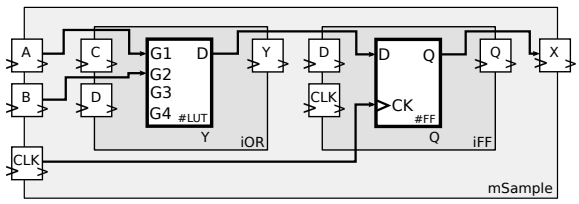

Fig. 6: Schematic view of the physical design model after the structural mapping phase (sample circuit)

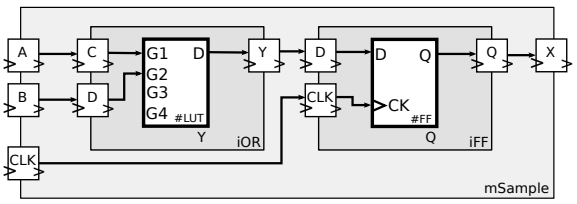

Fig. 7: Schematic view of the physical design model after the signal mapping phase (sample circuit)

extended parser automatically generates a module tree of given source files by using static (resolving module structure, instances, statements) and dynamic (parameterization, generate-statement execution) evaluation. The root node represents the top module, subsequent nodes are module instances. Each node contains *declarations (ports, wires, integers)*, *child nodes* (instances) and *port mappings* between its declarations and ports of child nodes. The structural information of the module tree is necessary for the PDM in the mapping phases.

Figure 5 illustrates the module tree of the simulation netlist of the sample circuit.

E. Structural Mapping Phase

In this phase, the structural information of the module tree is applied to the PDM. Therefore, we introduced the concept of *circuits* in the PDM. A circuit is a named module, which has several input, inout, and output ports. Each circuit has *primitives* and *child circuits*, which are in turn circuits with ports, primitives and child circuits, too. Initially, there exists only one circuit, which we call *root circuit*. It corresponds to the top module of the circuit description and initially contains all primitives.

In order to statically map elements from the circuit description to elements from the design, we need to transfer the structure of the circuit description to the PDM. Therefore it is necessary, to create a circuit for each node in the module tree as well as a circuit port for each node port. The available primitives are moved to the circuits, guided by the module tree specification.

Furthermore, we need to analyze the availability of a static mapping for each port, defined by the existence of the specified sources and sinks in the PDM. Only, if all sources and sinks exist, the circuit port is marked *mappable*, otherwise it is *unmappable*. There are circuit ports, that are expected to be unmappable: fixed-0 and fixed-1. Both are unnamed in the design and in the simulation netlist, located in the root circuit and cannot be mapped unambigously. That is why ports, which are connected to them in the simulation netlist, are expected to be unmappable. In order to avoid the creation of a high number of circuit ports due to their placement in the root circuit, fixed-1 and fixed-0 elements are copied to the circuits of their sinks. So, fixed-0 and fixed-1 elements lead to higher numbers of unmappable ports, but not to the creation of new circuit ports. In contrast, unexpected unmappable ports cause the creation of new circuit ports, because primitives, that are located in different circuits, need to be connected through circuit ports.

Algorithm 1 illustrates the structural mapping phase. The result of it is depicted for the sample circuit in Figure 6.

Algorithm 1 Structural Mapping Phase

```
1: procedure STRUCTMAP(Circuit circuit, Node node, PDM pdm)
2:     Map circuit ⟷ node;
3:     for all node ports nport in node do
4:         Create circuit port cport in circuit;
5:         Map cport ⟷ nport;
6:     end for
7:     Assign each Primitive prim (LUT, MUX,...) in pdm to a Circuit
8:     circuit by name matching(prim name starts with circuit name);
9:     for all Node nchild in childs of node do
10:        Create Circuit cchild with name of nchild below circuit;
11:        structMap(cchild, nchild);
12:    end for
13:    Move grounds and supplies to sinks circuits;
14:    Identify and remove unmappable ports;
15: end procedure
```

F. Signal Mapping Phase

In this phase, the signal flows (wires and port mappings) of the module tree are transferred to the PDM, based on the previously applied structural information. Algorithm 2 outlines the signal mapping phase. Algorithm 3 describes the transformation of sources and sinks from the simulation netlist to ports of the physical design model.

The fully mapped PDM of the sample circuit is illustrated in Figure 7.

Algorithm 2 Signal Mapping Phase

```
1: procedure SIGNALMAP(Circuit circuit, PDMpdm)
2:     for all circuit ports cport in circuit do
3:         NodePort nport ← Mapping of cport;
4:         PDMPort lSrc ← pSRC(leaf* source of nport);
5:         PDMPort dSrc ← pSRC(direct+ source of nport);
6:         PDMPort[] lSnk ← pSNK(leaf sinks of nport, lSrc);
7:         PDMPort[] dSnk ← pSNK(direct sinks of nport, lSrc);
8:         for all CircuitPort sink in lSnk do
9:             Remove wire from dSrc to sink;
10:        end for
11:        Create wire from dSrc to cport;
12:        for all CircuitPort sink in dSnk do
13:            Create wire from cport to sink;
14:        end for
15:    end for
16:    for all child circuits child in circuit do
17:        signalMap(child);
18:    end for
19: end procedure
20: PDMPort: port inside the PDM (circuit port or primitive port)
21: *leaf: node of node port has no child circuit
22: +direct: node of node port may have child circuits
```

G. Static Mapping Library

The presented flow results in a physical design model, that integrates modules and ports from the circuit description.

Algorithm 3 Translation of Node Ports to Ports in the PDM

```
1: procedure PSRC(NodePort nport): PDMPort
2:     if node of nport is not leaf then
3:         Node node ← node of nport;
4:         Node parent ← parent node of node;
5:         Circuit circuit ← Mapping of parent
6:         Primitives prims ← primitives in circuit
7:         primitive ← primitive in prims with name of node
8:         PrimitivePort[] outputs ← ports of primitive
9:         if outputs.length == 1 then return outputs[0];
10:        else return port in outputs with name of nport;
11:        end if
12:    else return (CircuitPort)Mapping of nport;
13:    end if
14: end procedure
15:
16: procedure PSNK(NodePort[] p, PDMPort lfSrc): PDMPort[]
17:    PDMPort[] result ← new PDMPort[];
18:    for all NodePort nport in p do
19:        Node node ← node of nport;
20:        if node of nport is not leaf then
21:            CircuitPort cport ← Mapping of nport;
22:            result.add(cport);
23:        else
24:            Node parent ← parent of node;
25:            Circuit circuit ← Mapping of parent;
26:            Primitive[] prims ← primitives in circuit;
27:            Primitive pri ← primitive in prims with name of node;
28:            for all input ports primInput in primitive do
29:                if primInput is connected to lfSrc then
30:                    result.add(input);
31:                end if
32:            end for
33:        end if
34:    end for
35: end procedure
```

For each port of the circuit description, it is possible to identify the physical ressources in the PDM, that are directly connected. Furthermore, each port provides the same logic signal in the PDM and the HDL, caused by preserving their structure throughout the design flow. Therefore, the PDM provides the bidirectional link between FPGA design and circuit description, that is useful for several applications – e.g. for the circumvention of FPGA design flow stages.

We implemented the proposed flow in a Java-based EDA tool library that we call *Static Mapping Library (StML)*. It utilizes a given FPGA design file (XDL) and its corresponding simulation netlist (VM), automatically creates the PDM and the module tree and executes the mapping phases. Finally, the StML returns the statically mapped physical design model, which may be further processed by external applications.

IV. Use Cases

In this section, we focus on two use cases of the StML. First, we introduce a location-based fault injection method, that uses the static mapping to circumvent synthesis, mapping, placing and routing. Afterwards, an approach for debugging circuits using FPGA-based emulation instead of simulation is proposed.

A. Fault Injection

Prototyping circuits using FPGAs has an important benefit: The execution speed is much higher compared to the achievable speed in simulations, while not incurring on high prototyping costs caused by hardware prototyping. However, prototyping does not only involve testing the circuit under common conditions. It further implies testing it in the occurence of faults in order to determine the behaviour of the faulty circuit and to partially improve its error-proneness.

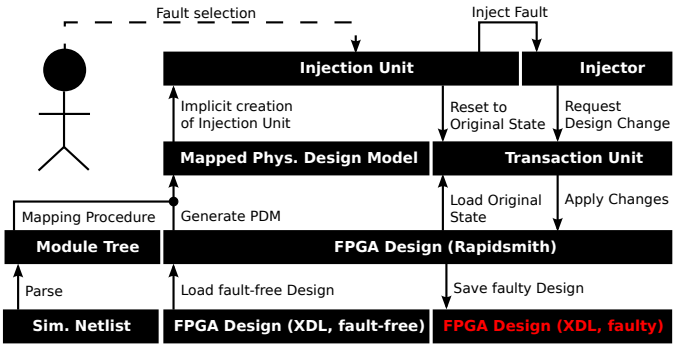

Fig. 8: Illustration of our fault injection infrastructure, built on top of the StML

That is why fault injection and fault emulation are important topics, providing a lot of existing approaches: (a) Instrumentation-based fault injection by adding further multiplexers and boolean gates to the primitives under testing [7][9]; (b) Fault injection by altering bitstreams [2] [8]; (c) Fault injection by altering the circuit description. [6]

Each approach has several benefits and drawbacks: (a) provides information about the fault location but enlarges the design size; (b) is very fast but provides no information about the fault location; (c) provides information about the fault location but is very slow due to running the synthesis and bitstream generation for each fault set.

In this section we introduce an approach for the injection of permanent faults. Utilizing the StML, we will inject faults directly into the FPGA design without losing the location of the faults refering to the HDL. In terms of fault analysis knowledge, the StML as well as (c) allow to draw inferences from the injection back to the HDL due to the bidirectional link. In contrast to (c), our approach circumvents the most time-consuming stages: Synthesis, mapping, placing and routing. Approach (b) does not allow to draw inferences back to the HDL and is therefore insufficient for improving the error-proneness of a circuit.

Figure 8 illustrates the infrastructure of our approach. The central parts are the *injectors*, assigned to primitive and circuit ports. Injectors implement the adjustments to instances, attributes or nets of the design. An injector, assigned to a port of a LUT, applies a change to its corresponding LUT equation. In contrast, an injector of circuit port only forwards its injection requests to connected primitive ports (because only primitives are physically existing in the design).

A fault list is passed to the *injection unit*, which invokes the injector of each fault location and causes changes to the design. Afterwards, the design written to an XDL file, needs to be converted to the NCD format and is passed to the bitstream generation in order to create the faulty bitstream. Finally, our method allows the injection of sets of permanent

	sample	s510	s38417	mips	cgrc
# ports	10	682	45010	2409	14211
# mapped ports	10	682	45010	2374	14058
% mapped ports	100	100	100	98,5	98,9
# deleted ports	0	0	0	35	153
# new ports	0	0	0	0	0

TABLE I: Mapping coverage of the tested circuits

faults into ports from the circuit description by using their correspondations in the PDM. It directly adjusts the XDL-based design and transfers the faults to the bitstream by a downstreamed execution of *bitgen*.

In a nutshell, we provide a fault injection method that circumvents synthesis, mapping, placing and routing without losing details about the fault location.

B. Debugging

Fault emulation, as described in the previous section, is only reasonable when there is a fully functional circuit implementation whose behaviour in the occurence of faults should be evaluated. A different use case for our static mapping becomes important in an earlier phase: Debugging. During the implementation phase, a developer may want to fix a bug in a circuit or a single component. For testing purposes, there are two obvious opportunities: Simulation and emulation. While an RTL-simulation allows the monitoring of specific signals, it lacks of the ability to detect more FPGA- and synthesis-related bugs. In contrast, FPGAs do not allow RTL-based monitoring without additionally synthesized components, but it exposes FPGA-related bugs.

Angepat et al. [1] describe an approach for an FPGA debugger that allows the tracking of registers without affecting the original HDL sources – they append a breakpoint controller beside, that pauses and continues the emulation run. When pausing a run, the bitstream is read back to a software platform using a JTAG interface, used to gather all register states. Afterwards, the design, bitstream and HDL are combined in a so-called *symbolic mapping*. The symbolic mapping is based on the Xilinx Logic Allocation File (.ll) and involves each existing register. It is further used to map the read back register states to the HDL code in order to infer existing bugs on the FPGA back to the HDL. Unfortunately, currently it is difficult to infer identified issues back to statements or blocks, because no other primitive than registers are mapped.

We address this issue by using our static mapping, which is not restricted to registers-only – it includes each primitive, that is documented in the Xilinx user guides (e.g. LUT or MUX). We gather the register values dynamically and we combine these values with the static mapping data. These further mapping details lead to more fine-grained debugging information. It can be used e.g. to analyze combinatorical logic, identifying a misbehaving LUT and inferring the wrong output signal back to the circuit description. Therefore our work enhances the approach of Angepat et al. with deeper static debugging knowledge.

	sample	s510	s38417	mips32	cgrc
# LUTs w/o A	1	107	2996	4081	8352
# LUTs with A	3	211	15866	4156	13384
% Area growth	200.0	97.2	429.6	1.8	60.2

TABLE II: Area overhead (number of LUTs) caused by disabled optimizations (code annotation)

V. Experimental Results

In this section, we illustrate the coverage of the static mapping as well as the area growth, in order to show the benefit and the tradeoff of our approach.

The experimental results have been gained using the toolchain, proposed in section II-A, with a Virtex-4 XC4VLX100 target device. We executed it on an Intel Core i5-3470 3.20GHz with 8GB of memory and Scientific Linux 6.3 (Kernel 2.6.32, 64-bit). We invoked the StML for several selected circuits: some ISCAS'89-benchmarks (gate-level), the sample circuit described in section II-D, a MIPS32-processor (RTL) and a coarse-grained reconfigurable core (CGRC, RTL).

A. Coverage of the Static Mapping

Table I outlines the number of ports for each tested circuit, which could be mapped successfully to the physical design model. The first row shows the absolute number of existing circuit ports. The subsequent rows outline the absolute and relative numbers of successfully mapped circuit ports. The fourth line determines the number of unmappable ports.

The number of created ports is especially interesting, because it gives a hint about the type of unmappable ports. As described in section III-E, fixed-0 and fixed-1 are expected to be unmappable. Therefore, they are moved to the circuits of their sinks and connected circuit ports are marked unmappable without creating new circuit ports. The results show, that there is no unexpectedly unmappable port – we have a full static mapping when disregarding grounds and supplies.

B. Area Overhead

Table II illustrates the growth of the area overhead caused by the prevention of synthesis optimizations due to the annotations described in section III-A (which are necessary for a static mapping). We measure the area overhead by the number of LUTs – the number of memory elements remained mostly the same, therefore it does not seem to be meaningful enough.

The first row determines the number of LUTs without annotating a circuit description, thus executing a normal design flow (this causes the design to be unmappable). The second row determines the number of LUTs when fully annotating each circuit description. The third row illustrates the relative area overhead that is caused by a full static mapping.

The results illustrate, that for RTL circuits the design growth is less than for gate-level ones. Especially the design growth of the s38417 attracts attention, but it can be explained easily. Due to the fine-grained modules (single and-,or-,xor-,not-gates,...) there are lots of synthesis optimization, which are prevented due to the annotations. In short: The more fine-grained the modules in a circuit description are, the greater

process duration (sec)	sample	s510	s38417	mips32	cgrc
design flow	64.77	70.14	169.73	181.66	344.82
xdl -ncd2xdl	3.35	3.38	10.26	5.34	9.95
mapping procedure	4.19	4.56	47.91	11.82	27.71
xdl -xdl2ncd	3.73	3.77	13.96	6.57	13.76
bitgen	21.16	21.62	30.01	26.83	39.47

TABLE III: Time consumption (seconds) of the design flow and the processes that participate in our mapping procedure (The first line describes the duration of one design flow run. Rows 2-4 describe onetime processes in our flow, while 5-6 describe multiple time executing processes.)

the design growth will be. If there is no need for a full annotation of a circuit description, it is possible, to annotate only those modules, a static mapping should be provided for, thus partially enabling optimizations and decreasing the area overhead.

C. Potential of Time Saving

Table III illustrates the durations of the standard design flow compared to the ones in our proposed flow – so it compares applying small design changes to the circuit description (standard) and passing the whole design flow with applying changes directly to the design, using the static mapping. This applies especially to the fault injection. The design flow for the larger tested circuits – for instance cgrc – takes 344 seconds. Injecting faults into the circuit description hence would cause a delay of 344 seconds for each fault set.

In contrast, our proposed flow passes the design flow once and has to execute some more initial processes (design conversion, mapping procedure). Afterwards, only design conversion and bitstream generation for each applied fault injection set has to be done, which leads to a delay which is less than the delay caused by the design flow.

Although our method causes an overhead for one execution. it saves a lot of time due to multiplely bypassing synthesis, mapping, placing and routing. For the cgrc, for instance, there is a time saving factor ≥ 6 from the fault injection to the generated bitstream (whole design flow versus design conversion + bitstream). This time saving factor is limited by the bitstream generation, which is the most time-consuming part in the current flow.

VI. Conclusion

In this paper, we illustrated an approach for the construction of a bidirectional link between an HDL source and its FPGA design. The bidirectional link regenerates the information of the relation between an FPGA design and its HDL circuit description, which is hidden during the FPGA design flow.

We implemented the mapping procedure in the Java-based EDA tool library. We further extended it with a system for the injection of permanent faults into the design while preserving the RTL-based location. It enables the selection of faults using the RTL description, but injects them directly into the FPGA design. We illustrated that our approach can be combined with existing approaches to support FPGA-based debugging. Therefore, our static mapping is able to accelerate and facilitate several kinds of use cases in the FPGA circuit development.

The experimental results support the feasibility of this approach and illustrate a coverage of the mapping procedure between 98.5% to 100.0% when involving grounds and supplies – or 100.0% when neglecting them. The results further show, that there is a tradeoff between the area overhead and the mapping granularity. In order to decrease the area growth, annotations for non-interesting modules may be left out, which leads to enabled compiler optimizations, causing less area overhead. Thus, the more coarse grained the annotations are, the less is the area overhead.

References

[1] H. Angepat, G. Eads, C. Craik, and D. Chiou. Nifd: Non-intrusive fpga debugger – debugging fpga 'threads' for rapid hw/sw systems prototyping. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 356–359, 2010.

[2] L. Antoni, R. Leveugle, and B. Feher. Using run-time reconfiguration for fault injection in hardware prototypes. In *Defect and Fault Tolerance in VLSI Systems, 2000. Proceedings. IEEE International Symposium on*, pages 405–413, 2000.

[3] C. Beckhoff, D. Koch, and J. Torresen. The xilinx design language (xdl): Tutorial and use cases. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pages 1–8, 2011.

[4] F. Benz, A. Seffrin, and S. Huss. Bil: A tool-chain for bitstream reverse-engineering. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 735–738, 2012.

[5] D. Brand, A. Drumm, S. Kundu, and P. Narain. Incremental synthesis. In *Computer-Aided Design, 1994., IEEE/ACM International Conference on*, pages 14–18, 1994.

[6] L. Burgun, F. Reblewski, G. Fenelon, J. Barbier, and O. Lepape. Serial fault emulation. In *Design Automation Conference Proceedings 1996, 33rd*, pages 801–806, 1996.

[7] K.-T. Cheng, S.-Y. Huang, and W.-J. Dai. Fault emulation: A new methodology for fault grading. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(10):1487–1495, 1999.

[8] D. de Andres, J.-C. Ruiz, D. Gil, and P. Gil. Fades: a fault emulation tool for fast dependability assessment. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 221–228, 2006.

[9] C. Dunbar and K. Nepal. Fault emulation and test pattern generation using reconfigurable computing. In *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pages 797–800, 2010.

[10] D. Gajski and R. Kuhn. Guest editors' introduction: New vlsi tools. *Computer*, 16(12):11–14, 1983.

[11] S. Guccione, D. Levi, and P. Sundararajan. Jbits: A java-based interface for reconfigurable computing.

[12] L. Kafka, M. Danek, and O. Novak. A novel emulation technique that preserves circuit structure and timing. In *System-on-Chip, 2007 International Symposium on*, pages 1–4, 2007.

[13] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs. In *Proceedings of the 21th International Workshop on Field-Programmable Logic and Applications (FPL'11)*, September 2011.

[14] J. Note and E. Rannaud. From the bitstream to the netlist. In *16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, page 264, 2008.

[15] R. K. Soni, N. Steiner, and M. French. Open-source bitstream generation. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 105–112, 2013.