Transparent In-Circuit Assertions for FPGAs

Eddie Hung, Tim Todman, Wayne Luk
Department of Computing
Imperial College London, UK
{e.hung,timothy.todman,w.luk}@imperial.ac.uk

***Abstract*—Commonly used in software design, assertions are statements placed into a design to ensure that its behaviour matches that expected by a designer. Although assertions apply equally to hardware design, they are typically supported only for logic simulation, and discarded prior to physical implementation. We propose a new HDL-agnostic language for describing latency-insensitive assertions and novel methods to add such assertions transparently to an already placed-and-routed circuit without affecting the existing design. We also describe how this language and associated methods can be used to implement semi-transparent exception handling. The key to our work is that by treating hardware assertions and exceptions as being oblivious or less sensitive to latency, assertion logic need only use spare FPGA resources. We use network-flow techniques to route necessary signals to assertions via spare flip-flops, eliminating any performance degradation, even on large designs (92% of slices in one test). Experimental evaluation shows zero impact on critical-path delay, even on large benchmarks operating above 200MHz, at the cost of a small power penalty.**

I. Introduction

Field-programmable gate arrays (FPGAs) are a general-purpose silicon technology capable of implementing almost any digital design. This prefabricated flexibility provides generic logic resources (e.g. lookup-tables and switched routing interconnect) that can be configured at implementation-time. Synthesising a design onto an FPGA uses Computer-Aided Design (CAD) tools to compute a feasible configuration of a subset of these resources to implement the requested circuit.

Modern FPGA devices can exceed 20 billion transistors; hence, (i) FPGA CAD can be time-consuming [1], and (ii), due to the heuristic nature of CAD algorithms, synthesised solution quality can be unstable. Rubin and DeHon [2] find small perturbations to initial conditions of routing algorithms affect delay by 17–110%. Thus, circuit modifications require resynthesising — a lengthy procedure, which may return worse results and impact designer productivity.

We present a solution inserting new, *latency-oblivious*, logic, such as in-circuit assertions, into an existing design transparently without needing to recompile the entire circuit. We define a latency-oblivious circuit to contain no strict constraints on the number of clock cycles for computing its result; one example is using trace-buffers to record on-chip signal behaviour [3]: pipelining trace signals does not affect observability. Another example is invoking circuit reset when the system becomes unresponsive. The key advantage of latency-oblivious circuits is that they introduce a new dimension of synthesis flexibility, allowing transparent insertion.

Traditionally, digital circuits have been developed using a logic simulation environment due to unlimited signal visibility, fast recompilation cycles, and software-like instrumentation. However, as designs become increasingly complex, circuit simulation speed slows. In turn, this causes circuit testing to be less thorough, and reduces designer productivity.

A promising approach uses *in-circuit assertions* [4] to verify designs at run-time. Because they run in the same circuit as the design under test, in-circuit assertions can run much faster than simulation, allowing testing to be more thorough. In-circuit assertions can be latency-oblivious since designers typically care more about if any assertions were violated rather than needing to be alerted immediately.

We insert additional logic, such as assertions, *transparently*, without affecting performance or functionality. We therefore insert post place-and-route, using only spare FPGA resources not used by the original user circuit. By using such *mutually exclusive* resources, new functionality can be added without affecting the user design. To eliminate any impact on the critical-path of the original design, we aggressively pipeline the new circuitry, enabled by its latency-oblivious nature. Our methods allow even large circuits to be thus augmented – we have tested on circuits using up to 92% slices of a large FPGA. We thus make the following contributions:

- An approach for reclaiming the spare, unused, resources on FPGAs to transparently insert new logic such as in-circuit assertions after circuit implementation.
- An assertion language based on Boolean logic allowing assertions to be described at high level.
- Use of minimum-cost graph flow techniques to simultaneously pipeline-and-route all input signals required by this logic, without impact on circuit timing.
- Extending to in-circuit exceptions, allowing some circuit errors to be fixed without rerunning place-and-route.
- Experimental validation, showing that our techniques incur only a small power penalty.

The remainder of this paper is organised as: Section II reviews related work, Section III shows our assertion language; Section IV describes our transparent insertion approach in detail. Section V describes exception handling. Sections VI and VII evaluate the methodology and show experimental results. Finally, Section VIII concludes, outlining future work.

The key concepts in this manuscript were first presented in [5]. Since then, we have developed a new language to describe latency-oblivious assertions and exceptions. The high-level language allows compact description of complex assertions, and translation to multiple design descriptions.

II. Background and Related Work

Latency-insensitive design: We exploit the flexibility of inserting latency-oblivious logic — logic without strict constraints on the number of clock cycles in which it must return a result. An example of latency-oblivious logic is trace-buffers used to record on-chip signal activity for debugging; pipelining each traced signal does not affect its observability. Latency-insensitive design [6] is a methodology to create designs that are insensitive to communication delays between components, allowing tools to pipeline them arbitrarily to meet performance criteria. This improved flexibility comes at the cost of area overhead and is unsuited to designs with poor communication locality. Note that only the elements we add are latency-insensitive: the rest of the design need not be.

In-circuit assertions: Assertions specify boolean conditions that should always hold true if the design is working correctly. An example in software may be that a 'malloc()' system call must return a non-zero value; a hardware example could check the carry-out bit of an adder is always '0' to indicate no overflow occurs. While it may not be practical to halt a hardware prototype in the same way as in simulation, it is nonetheless beneficial to alert the designer if any assertion fails. Assertions may be combinational, or could include state as well, for example, checking that each DRAM access latency lies within a bound, or even statistical properties [7].

Hardware assertions form part of the SystemVerilog language standard (SVA) [8], and can also be described using the Property Specification Language (PSL) [9]. Typically, such constructs are supported only by logic simulators or formal verification tools and are discarded for hardware, although researchers have proposed extending these into silicon [4], [10]. Previous approaches, however, insert assertions by modifying the original hardware description and resynthesising the entire circuit — HLS assertions can degrade FPGA performance by 3% [4]. Although incremental compilation approaches can accelerate this procedure, commonly the original circuit must be partitioned in advance to reserve space for assertions.

Network flow algorithms in FPGA tools: A *flow* network is a graph $G(V, E)$, with a set of vertices V and a set of directed edges E, each edge connecting two vertices and with capacity $u \in \mathbb{N}$. A valid flow solution exists when (i) the flow carried by each edge does not exceed its capacity, and (ii) conservation of flow exists at all vertices — the sum of all flows entering a vertex must equal the sum of all flows exiting — with two exceptions at the source and the sink. The source node may only produce flow; the sink node may only consume flow. A single-commodity network has only one type of flow present.

Efficient algorithms to compute the maximum integer flow of a single-commodity network exist (multi-commodity maximum integer flow is known to be NP-complete), and are applied in FPGA CAD. FlowMap [11] employs a max-flow algorithm (specifically, its dual, the min-cut) during FPGA technology-mapping to compute a mapped netlist with the minimum logic-depth, while Lemieux et al. [12] use max-flow to evaluate routability of depopulated FPGA switch-matrices.

Combining both min-cost and max-flow algorithms is reference [3], where they are used to connect signals to trace-buffers during FPGA debug. In contrast, we use flow techniques in this work to concentrate signals into a single region (rather than connecting to trace-buffers distributed across the device) in a way that does not impact the circuit performance. While prior work reports that adding trace-buffer connections reduced the maximum clock frequency from 75MHz to 55MHz, we pipeline our signal routing to mitigate all impact on performance.

Recent work on incremental trigger insertion [13] uses spare FPGA resources to insert trigger circuits for enabling debug buffers. Unlike our approach, this work incurs critical path delay penalties up to 107%, due to not pipelining the signals.

III. Assertion Language

We develop a high-level language for describing in-circuit assertions, based on Boolean logic, and show an implementation by systematic translation into a target language such as VHDL. Since assertions are written in a high-level language, they are independent of particular implementations, thus potentially reusable between different but related designs.

Compared to industrial assertion languages such as SVA, our assertion language corresponds to SVA's concurrent assertions, which evaluate once per cycle and run concurrently with design code. Our language does not support SVA immediate assertions, since these depend on simulation concepts such as delta time. Unlike SVA, our assertions are not limited to VHDL designs, but can target other descriptions such as Verilog and OpenSPL.

The assertion language includes useful primitives for complex designs: arithmetic expressions including floating-point, counters and accumulators, allowing complex assertion conditions without needing to use lower-level primitives as in VHDL. Delays allow assertions to match latencies of pipelined circuits. Users can declare external hardware blocks, allowing assertion conditions to use design-specific primitives.

An extended Backus-Naur form grammar follows (A^* denotes zero-or-more repetition, $A^?$ denotes optional items, bold text denotes keywords, capitals denote literals):

$$\begin{aligned}
d =\ & \mathbf{user}\mathrm{ID}(<e(,e)^*>)^?(e(,e)^*)(\{\mathbf{latency}=e\})^? \\
\mid\ & \mathbf{assertion}\mathrm{ID}(<e(,e)*>)^?(t\mathrm{ID}(,t\mathrm{ID})^*)\{s;(s;)^*\} \\
t =\ & \mathbf{int}<e> \mid \mathbf{uint}<e> \mid t[e] \\
s =\ & \mathbf{var}ID = e
\end{aligned}$$

$$\begin{array}{rllll}
e = & \mathbf{true} & \mid\ \mathbf{false} & \mid\ \mathrm{INT} & \mid\ \mathrm{FLOAT} \\
\mid & e \bullet e & \mid\ e \circ e & \mid\ -e & \mid\ \neg e \\
\mid & [e(:e)^?] & \mid\ \mathbf{accum}(e,e) & & \\
\mid & \mathbf{counter}(e,e) & \mid\ \mathbf{delay}<e>(e) & &
\end{array}$$

where d are declarations, including user-defined blocks which can be used in assertions, with static (generic) parameters in angle brackets and run-time parameters in round brackets and optional latency specification, or assertion declarations also with static and typed generic parameters; t represents types with declared bitwidths – only signed and unsigned integers and arrays of these are supported; s assigns an expression to a variable – once assigned, a variable acannot be reassigned; e declares expressions: $\circ \in \{+, -, *, /, <, \leq, >, \geq\}$ (arithmetic expressions with their usual priority); $e_1 @ e_2$ bit-concatenates e_1 to e_2; INT and FLOAT respectively represent integer and floating-point literals; $e_1[e_2]$ selects a single bit or an array element at position e_2 depending on the type of e_1;

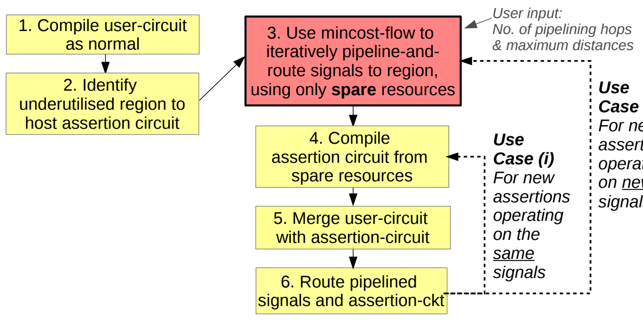

1. Compile user-circuit as normal
2. Identify underutilised region to host assertion circuit
3. Use mincost-flow to iteratively pipeline-and-route signals to region, using only spare resources
User input: No. of pipelining hops & maximum distances
4. Compile assertion circuit from spare resources
5. Merge user-circuit with assertion-circuit
6. Route pipelined signals and assertion-ckt
Use Case (i) For new assertions operating on the same signals
Use Case (ii) For new assertions operating on new signals

Fig. 1: Transparent assertion logic insertion approach.

$e_1[e_2 : e_3]$ selects bit range e_2 to e_3 inclusive; $\mathbf{accum}(e_1, e_2)$ accumulates values of expression e_1, resetting to zero when e_2 is true; $\mathbf{counter}(e_1, e_2)$ counts repeatedly from e_1 to e_2 in single steps; $\mathbf{delay} < e1 > (e2)$ delays $e2$ by $e1$ cycles.

Translating to latency-oblivious assertions: We systematically translate from the assertion language into a latency-oblivious implementation. The translation is syntax-directed, proceeding recursively from the root of the assertion condition to the leaves, which will be atomic propositions or Boolean literals (true or false). Each operator maps one-to-one to a block in the implementation – for example, to a VHDL block implementing that operator. The only restriction is that the latency in cycles of the resulting circuit must be the same from each circuit input to the circuit output, ensuring all data is synchronised. The circuit can be arbitrarily pipelined to meet the timing of the design under test. We automatically insert pipeline registers to ensure inputs from the same cycle arrive on the same cycle throughout the graph using a straightforward ASAP (as soon as possible) algorithm.

Example: An assertion checking signal C is in range $[L, H]$:

```
1 assertion inRange<L, H>( uint<32> C)  {
2   (L <= C) ∧ (C <= H);  }
```

where line 1 declares an assertion with two compile-time parameters `L` and `H` (inside the angle brackets) and one run-time parameter `C`; line 2 is an expression checking that `C` is in the range [`L`, `H`]. This could be used wherever a value must be in a defined range, for example to ensure a soft CPU only reads instructions from a valid memory space.

IV. Transparent Logic Insertion

Figure 1 shows our approach to inserting new logic transparent circuitry in six steps: *Step 1* compiles the user-circuit as normal (for example, by using Xilinx ISE) without reserving any resources a-priori or specifying additional constraints over a regular compilation run. *Step 2* examines the floorplan of the compiled result, identifying an underutilised region (typically at the peripheries of the device) that could host any new logic. Currently, this step is manual; future work could automate it.

Step 3 applies minimum-cost flow techniques to transport user signals (perhaps distributed across the whole device) needed by the assertion circuit into its vicinity, via pipelining registers. The exact number of pipeline stages, and the maximum distance between stages are user parameters. Crucially, only spare logic and routing resources not consumed by the original circuit are used — this makes our approach transparent.

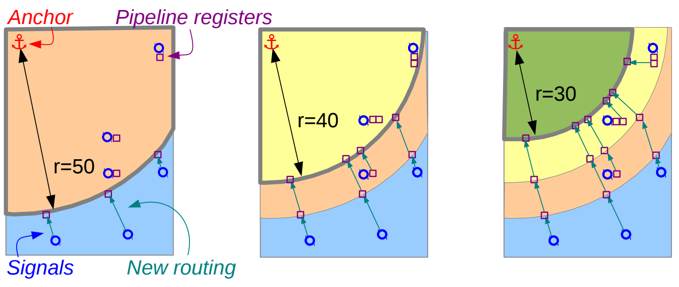
Based on results from *step 3*, which specifies a template containing the location of all flip-flops used in pipelining, and all logic resources occupied by the user circuit, *step 4* applies vendor tools to compile (but not route) a separate circuit implementing the new logic tailored to this template, again using only those spare resources. As this new logical circuit is mutually exclusive to the original user circuit, *step 5* merges the pipelined-and-routed circuit from *step 3* with the newly placed circuit from *step 4*. Finally, *step 6* completes the unrouted connections inside the merged circuit (connecting from the final pipelining stage to the new circuit, and within the new circuit) using vendor tools.

For new functionality using the same set of pre-routed signals $\big($*case (i)* of Fig. 1$\big)$ only steps 4 to 6 would need to be repeated. However, for new logic operating on signals not already routed $\big($*case (ii)*$\big)$ step 3 must also be repeated, to compute new pipelined connections for any new signals.

Pipeline-and-route: A key ability of this toolflow is transporting circuit signals, perhaps scattered across a device, into a concentrated region as inputs to a new circuit, while only using spare resources. Routing such signals directly incurs large distance-dependent routing delays. To mitigate these delays which can introduce new critical-paths, we pipeline the signals. As our approach targets latency-oblivious logic, additional pipelining stages are acceptable. Although fanout increases by one for each signal routed, this is unlikely to affect overall design timing. Modern commercial FPGAs contain buffered routing – adding an extra routing branch to an existing net incurs only a small capacitative load; on the Xilinx platform we use in testing, timing analysis reports the effect as < 5ps.

We transform the FPGA routing resource graph (with nodes occupied by the user circuit removed) into a flow network using similar techniques to [3] and employ minimum-cost flow techniques to route all necessary signals to unique pipelining registers from a candidate set. An important degree of freedom with this particular routing problem (and that does not exist with user routing) is that each signal can connect to any register from the candidate set; this provides significant routing flexibility even under constrained scenarios. Our approach differs from the separate placement and routing stages employed by traditional CAD tools; in some ways, our tool can be seen as routing signals, resolving congestion, and placing pipelining registers *simultaneously*. Furthermore, unlike reference [3], we do not seek to find the routing solution with maximum signal observability, but instead use flow algorithms to perform both placement and routing during signal pipelining.

Given timing estimates (costs) for each edge in the flow network, the objective function minimised is the *average-case* timing for each connection — not the *worst-case* timing across all connections determining the critical-path delay. Nevertheless, our experiments show that when a user chooses the candidate register set conservatively (via the number of pipelining hops, and the distance of each hop from the anchor point), our approach can return solutions that do not increase critical-path delay. It is worth pointing out that we do not apply min-cost flow techniques to find the optimal timing solution, for the



(a) First hop (b) Second hop (c) Third hop

Fig. 2: Pipeline-and-route technique — by iteratively decreasing the set of candidate registers (as outlined, specified using radius r) from anchor point, signals are pipelined to their destination.

following reasons: *a*) due to the nature of the network flow problem, it is only possible to optimise for average-case timing, *b*) we modify the network heuristically to guide algorithm behaviour in ways that do not reflect the true device, and *c*) while each application of min-cost flow is proven to find the global optimum, when applying this technique iteratively (in a piecewise fashion) to each pipeline stage, optimality is no longer guaranteed. Instead, we consider the flow approach to be an effective heuristic for this particular routing problem.

In our tool, the candidate set of registers is specified as spare flip-flops that fall within a user specified radius from an (X, Y) anchor location. Spare flip-flops may exist inside slices partially occupied by the user circuit (care must be taken to ensure that such logic slices belong to a compatible clock domain to the signals being transported) or within unoccupied sites. The region determined by the anchor and radius is a circle (or a segment, if clipped by the FPGA boundary). By iteratively reducing the radius of this circle over multiple routing passes, and hence reducing the candidate set of pipelining registers, it is possible to migrate signals to the anchor point, at the cost of additional latency for each pipeline hop. Figure 2 illustrates: in each iteration, signals outside of the candidate region are routed into its minimum-cost flip-flop inside the region. Those signals already inside the region are routed to a different flip-flop inside the region, to maintain latency between signals.

To guide the min-cost flow algorithm towards a valid routing solution, we make two heuristic modifications to our network. Firstly, we apply a penalty to all network edges crossing FPGA clock regions. In most devices, all resources are exclusively associated with a single clock region, and due to the clock network design, signals crossing between regions incur clock skew. In our experiments, we observe that sometimes the min-cost algorithm returns very short routing paths bridging across two different regions, which combined with a positive clock skew, result in a hold time violation. To discourage such paths, we add an inflated delay penalty to all such edges. Secondly, we observe that it is possible for the min-cost algorithm to connect to pipelining registers whose output pin is blocked due to routing congestion. Given that we route signals piecewise, it would not possible for one min-cost iteration to understand the routeability of the next iteration. To alleviate this, during candidate flip-flop selection, we prune all registers without sufficient free fan-outs left for downstream usage.

V. Exceptions: semi-transparent logic insertion

Our method inserts logic transparently (circuit behaviour is preserved), but there are limits to what transparent insertion can achieve; essentially, we are limited to adding extra circuit outputs. In this section, we extend our method to allow limited changes to circuit behaviour (abandoning strict transparency), which can allow faults in the circuit to be corrected. By analogy with software, we call these additions *exceptions*; like software exceptions they allow error correction and recovery. We call these additions *semi-transparent*: they only affect the replaced circuit signal; the rest of the circuit is not directly affected.

Motivating example: the previous section uses range checking: detecting that a circuit signal, such as the program counter in a soft-processor, lies within a valid range. Correcting the program counter could, for example, replace a faulty value with the address of a service routine, allowing operating system software to handle the error. Our approach applies to exceptions where a bounded amount of latency can be tolerated — such as the program counter for a pipelined processor.

Assertion language extensions for exceptions: We extend our assertion language to allow for exceptions. Unlike software exceptions, each exception maps one-to-one with an assertion. An extended Backus-Naur form grammar follows:

$$d = \cdots \mid \ \textbf{assertion}\mathrm{ID}(<e(,e)*>)^{?}(t\mathrm{ID}(,t\mathrm{ID})^{*})\{s;(s;)^{*} \\ \textbf{catch}\{\mathrm{ID} = e\}\}$$

where the extra production allows an assertion declaration to have an exception handler: an assignment statement, allowing one of the run-time arguments to the assertion to be overwritten.

The program counter range-checker could look like:

```
1 assertion inRange<L,H,OutOfRangeTrap>(uint<32> C) {
2   (L <= C) ∧ (C <= H);
3   catch {
4     C = OutOfRangeTrap;   } }
```

where lines 1-2 declare the assertion as before; lines 3-4 declare an exception handler which, if the assertion is triggered, overwrites the circuit signal C with the value `OutOfRangeTrap`, the address of the software handler, if the assertion fails.

Implementation: Figure 3 shows our implementation using the methods of Section IV: `SRC` is the circuit signal with associated assertion and exception. First, inputs to assertion and exception logic are transported to the spare (possibly disjoint) logic region(s), where the assertion and exception are located. Next, we apply the pipeline-and-route method again to transport the assertion condition and exception value back to its original driver. A multiplexer chooses between `SRC` and the exception value depending on the assertion condition, re-using as much routing as possible. Note the total latency in cycles from `SRC` to `SRC'` via the exception path will be the sum of the latency through exception and assertion circuits (which must be equally balanced), and may be required to be less than some constraint (e.g. the processor pipeline depth). Although the two circuits must be balanced, this need not be to a fixed value; furthermore, this latency can be arbitrarily distributed between input and output links and in the spare region, for further CAD flexibility.

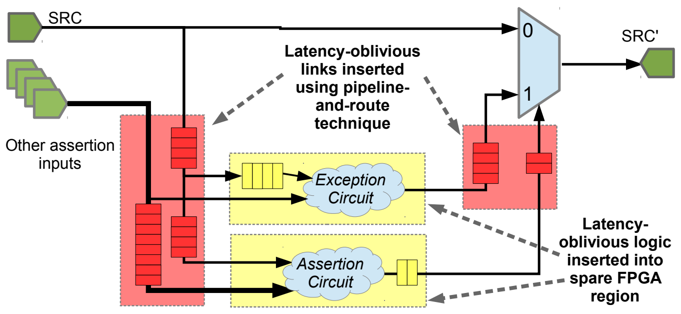

SRC
SRC'
0
1
Latency-oblivious links inserted using pipeline-and-route technique
Other assertion inputs
Exception Circuit
Assertion Circuit
Latency-oblivious logic inserted into spare FPGA region

Fig. 3: Exception architecture: both assertion and exception are located in spare logic regions. We apply pipeline-and-routing twice: (1) to transport assertion and exception inputs to their respective circuits; (2) to transport the assertion condition and exception values back. A multiplexer overwrites original signal `SRC` with the exception value if the assertion is triggered.

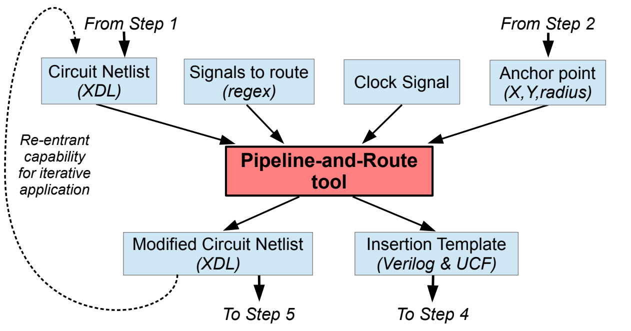

From Step 1
From Step 2
Circuit Netlist (XDL)
Signals to route (regex)
Clock Signal
Anchor point (X,Y,radius)
Re-entrant capability for iterative application
Pipeline-and-Route tool
Modified Circuit Netlist (XDL)
Insertion Template (Verilog & UCF)
To Step 5
To Step 4

Fig. 4: Our custom pipeline-and-route tool used in step 3.

Semantics and correctness: Clearly adding exceptions is not completely transparent, since circuit behaviour is changed if an assertion with an exception is violated. However, circuit timing can be preserved if 1) the monitored signal `SRC` is not on a critical path and 2) the additional multiplexer does not make the path from `SRC` to its downstream readers critical.

Even if timing is preserved, the circuit could still be incorrect: if the assertion is triggered, the monitored signal is delayed by several cycles, because it passes through the pipeline-and-route network and the assertion and exception logic. For some applications, this will not matter: the datapath of video or audio applications may tolerate a few cycles of delay. In the program counter example, the processor runs for a few cycles (but before any erroneous computation is flushed) before jumping to the trap routine: the exception value replaces the program counter.

Summary: extending our approach to allow exceptions, replacing a circuit signal by an exceptional value if the assertion is triggered, is not transparent, but can be useful in some applications if care is taken not to alter the critical path or introduce a new critical path. In general, the resulting circuit may not be identical; however, for some useful applications this approach allows a circuit to be corrected without rerunning the time-consuming place-and-route process.

VI. Evaluation methodology: Xilinx

Although we believe that our techniques can apply to all FPGA vendors, we evaluate our work on Xilinx technology. In our evaluation, we first employ Xilinx ISE v13.3 to compile the original user circuit (step 1 from Fig. 1). For designs with timing constraints we apply those to ISE, but for designs without we operate ISE in ‘performance evaluation mode’ which infers all clocks from the circuit and minimises their periods. For step 2, we open the compiled design in Xilinx’s FPGA editor to visualise its floorplan, and identify an underutilised region to host any new circuitry.

Next, step 3 translates the place-and-routed netlist returned by ISE from its proprietary binary format, NCD, into the Xilinx Design Language format, XDL using command `xdl -ncd2xdl`. The XDL format is human-readable and contains a complete description of Xilinx netlists: from LUT masks, component placements, to source and sink pins, and even which individual wires comprise every routed net. Toolkits, such as Torc [14], can manipulate this format.

After decoding the circuit, we apply our pipeline-and-route tool (using Torc to manipulate XDL, and LEMON [15] for flow computations) to execute the procedure described in Section II. Figure 4 illustrates: given an XDL circuit netlist, a set of signals to be routed (possibly regular expressions matching nets in the XDL netlist), their clock domain, and the set of candidate registers (specified by an anchor point (X, Y) and radius r) the tool applies our techniques to transport all signals to a pipelining flop within this region. The output is an augmented circuit netlist in XDL format, and a template that can be used to build the new circuit in the next step: a Verilog file specifying the location of all pipelining registers, and a Xilinx User Constraints File (UCF) specifying which resources on the device are occupied (using the `PROHIBIT` constraint). Our pipeline-and-route tool is re-entrant: the output netlist can be used as the input netlist for the next routing run, allowing this procedure to be executed iteratively for each pipeline hop.

Step 4 takes the template produced in the previous step, adds new functionality into the source, and synthesises and places (without routing) this circuit using ISE. The UCF constraints file forces: 1) mutual exclusivity between logic resources in user and the assertion circuits; 2) the Xilinx placer (with the `AREA_GROUP` constraint) to use only the host region identified in step 2. Note it is currently impractical (perhaps impossible in the Xilinx toolflow) to enforce mutual exclusivity on routing resources. For step 5, we translate the added circuit into XDL, then use a custom tool to merge with the circuit from step 3.

Finally, step 6 converts the merged XDL circuit into NCD format using command `xdl -xdl2ncd` (also invoking the Design Rule Check, DRC) and invokes the router in re-entrant mode to 1) route the added circuit, 2) complete last-mile routing from the final pipelining stage to the new circuit’s inputs. We set the `RCT_SIGFILE` environment variable to force use of only spare routing instead of ripping-up user circuit nets.

We target the Xilinx ML605 evaluation kit, containing a Virtex6 FPGA (xc6vlx240t) with 150,000 six-input LUTs within a grid of 162×240 slices. We employ four benchmarks, chosen for complexity and high clock rates: LEON3, a System-on-Chip design; two variants of an AES encoder/decoder; a floating-point datapath. For each, we insert assertions to verify correct operation.

	Exp. 1: LEON3 SoC		Exp. 2: AES (3 pair)		Exp. 3: AES (2 pair)		Exp. 4: FloPoCo	
	This work	Resynthesis	**This work**	Resynthesis	**This work**	Resynthesis	**This work**	Resynthesis
User circuit:								
Slice utilization	30,698 (81%)		34,880 (92%)		26,362 (69%)		24,650 (65%)	
LUT utilization	82,830 (54%)		108,132 (71%)		71,976 (47%)		61,967 (41%)	
Register utilization	60,725 (20%)		32,022 (10%)		21,391 (7%)		97,968 (32%)	
Critical-path delay	13.324ns		4.213ns		4.153ns		6.232 ns	
Pipe-and-routed ckt:								
Signals routed	240	-	384	-	512	-	144	-
Slice utilization	30,720 (+22)	-	34,985 (+105)	-	26,890 (+528)	-	24,790 (+140)	-
LUT utilization	82,925 (+95)	-	108,264 (+132)	-	72,216 (+240)	-	61,996 (+29)	-
Register utilization	61,205 (+480)	-	33,942 (+1,920)	-	23,951 (+2,560)	-	98,400 (+432)	-
Critical-path delay	13.324ns	-	4.213ns	-	4.153ns	-	6.232ns	-
Pipeline latency	2	-	5	-	5	-	3	-
Assertion circuit:								
Slice utilization	30,770 (+50)	33,642	35,140 (+155)	35,104	28,045 (+1155)	25,807	24.839 (+49)	23,842
LUT utilization	83,078 (+153)	82,489	108,831 (+567)	108,591	76,478 (+4262)	75,996	62,163 (+167)	63,738
Register utilization	61,454 (+249)	60,973	34,636 (+694)	32,689	28,385 (+4434)	27,765	98.550 (+150)	98.100
Critical-path estimate	3.729ns	-	2.436ns	-	2.758ns	-	3.162ns	-
Assertion latency	3	3	8	8	8	8	3	3
Final circuit:								
Critical-path delay	13.324ns	13.327ns	4.213ns	4.205ns	4.153ns	4.318ns	6.232ns	10.085ns

TABLE I: Detailed comparison between our proposed method and the resynthesis approach.

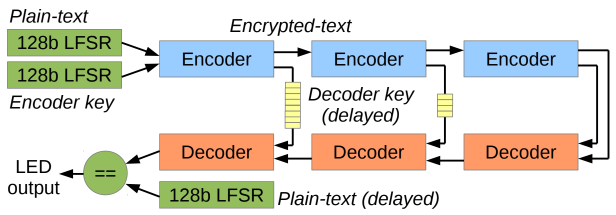


Fig. 5: Experiment 2: AES (3-pair) encoder+decoder.

Benchmark 1: LEON The Aeroflex Gaisler LEON3 [16] is an open-source VHDL multi-core SoC design capable of booting Linux, parameterised to customise the number, size and configuration of SPARC cores and on-chip peripherals. We configure the LEON3 with 8 cores, each with 64kB of I-cache and D-cache, and MMU, DDR3 memory controller, Ethernet and CompactFlash peripherals. The LEON3 ML605 template constrains the main SoC clock to 75MHz (13.33ns).

Benchmarks 2 and 3: AES For a datapath orientated benchmark, we build two variants of a 128-bit AES encoder/decoder; Fig. 5 shows a block diagram for the 3-pair variant. The circuit is derived from [17], modified to insert an extra pipelining stage in each AES round, improving performance but doubling encoding and decoding latency to 20 cycles. The advantage of this benchmark is that it is entirely self-stimulating (both plain-text and encoder key inputs generated by linear-feedback shift registers), and self-checking, with each encoder paired with a decoder allowing the decoded result to be verified against the original plain-text input (regenerated via an offset LFSR).

Benchmark 4: FloPoCo Lastly, we use a floating-point datapath built using FloPoCo [18]. We use P parallel copies of a W-tap single-precision floating-point moving average filter. Each filter's input is stimulated using one 32-bit LFSR; for a 400MHz target frequency, FloPoCo returns a circuit with pipeline latency of 45. To generate a medium utilisation circuit, we choose P=24, W=8 and disable shift-register extraction in ISE (which would convert pipeline registers to shift-registers), creating a benchmark with higher flip-flop utilisation.

VII. Results

A. Experiment 1 — simple in-circuit assertion for LEON3: We insert an assertion to check the program counter for each of 8 cores lies in the memory space of the memory controller, checking instructions only come from main memory.

Using our assertion language, the assertion is shown in Section III; we systematically translate this to the implementation.

Unmodified, the LEON3 benchmark consumes 81% of logic slices, 54% of LUT resources, meeting a 13.33ns (75MHz) clock constraint (Table I, column 2). Examining the floorplan shows an underutilised region by the upper-left of the device; the anchor point is (0,185). We invoke pipeline-and-route twice (step 3 from Fig. 1), transporting signals towards the anchor via two stages, with radii 160 and 80 respectively. In total, 240 bits are routed: the 30-bit program counter (the 2 least significant bits are unused) for each of the 8 cores. The resulting circuit consumes modest additional resources (registers from existing and new slices, plus LUTs used as route-throughs).

Next, we synthesise the assertion circuit (step 4); it occupies 50 slices and 153 LUTs over the pipelined circuit. Due to the simple assertion, the pre-routing critical-path timing estimate for its pipelined circuit is 3.73ns (in fact, the estimated critical-path is between the final pipeline stage and the assertion circuit), comfortably meeting the 13.33ns circuit constraint. After merging and routing the assertion circuit with the user circuit (steps 5 and 6) we find that no new critical-paths have been introduced, and the circuit meets timing at 13.32ns.

We compare the efficiency of our transparent logic insertion with the traditional approach of adding the assertion at source-level and resynthesising the whole circuit. To ensure fairness, we manually modify the source code to extract the signals of interest out through the circuit hierarchy, attaching them to an identical instance of the assertion HDL. Table I shows the results under the 'Resynthesis' heading. While the final result shows that, for this experiment, there is no impact on timing because both circuits meet the constraint, designers would still have to resynthesise their circuit for each set of assertions. Interestingly, there is a significant 10% difference in logic

Place seed → Benchmark ↓	#1 (ISE default)	#2	#3	#4	#5
AES (3x) user	4.338	4.418	4.374	4.515	**4.213**
AES (3x) resyn	4.929	4.387	4.635	4.279	**4.205**
AES (2x) user	4.252	4.497	4.301	4.666	**4.153**
AES (2x) resyn	4.917	4.678	4.468	5.240	**4.318**
FloPoCo user	6.542	9.408	9.877	**6.232**	9.891
FloPoCo resyn	9.892	**6.157**	9.723	10.085	10.719

TABLE II: Critical-path delay (ns) fluctuation under different placement seeds.

slice utilization between the original and instrumented circuits; apparently adding a small amount of extra logic causes the CAD tools to make very different packing decisions.

Figure 6 charts the runtime advantage of our approach. On this benchmark, inserting assertions transparently is 3.9 × faster than resynthesising. For pipeline-and-routing, runtime is dominated by final routing using vendor tools.

B. Experiment 2 — stateful assertion for AES (3-pair): Our second experiment inserts stateful assertion logic into a circuit with both high maximum clock frequency and high device utilization: AES, with 3 encoder-decoders pairs (Fig 5).

Using our assertion language, the assertion is:

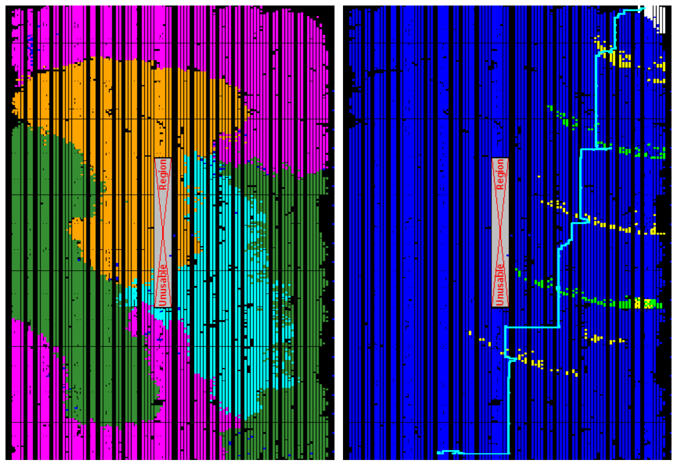
```
1 user uint<128> deAES( uint<128>, uint<128> )
2   { latency=N };
3 assertion checkAES( uint<128> msg, uint<128> key1,
4   uint<128> key2, uint<128> key3, uint<128> enc ) {
5   delay<4*N>( msg ) == deAES( delay<3*N>(key1),
6     deAES(delay<2*N>(key2),
7       deAES(delay<N>(key3), enc)) );  }
```

where lines 1-2 declare the AES decoder (a user-defined block `deAES` with latency `N`) and lines 3-7 define the assertion as a chain of AES decoders; delayed keys balance decoding latency.

This circuit uses 71% of the LUTs, 92% of logic slices, showing that our methods apply to large designs. The AES circuit has no timing constraints, so we operate ISE in performance evaluation mode to find the best timing; to mitigate CAD noise, we compile using five different placement seeds (placer cost tables), the best result returns a critical-path delay of 4.21ns, or 237MHz. Table II lists timing for all five seeds.

Examining the original circuit floorplan (Fig. 7a) we see the top-right region of the device is underutilised, and invoke our tool five times to pipeline-and-route signals into this region. We chose the top-rightmost coordinate as the anchor position (161,239), using decreasing radii on each iteration: 200, 160, 120, 80, 40. The signals we pick are 128-bit buses taken from each of the 3 encoders (specifically, the `key_out[127:0]` register from the fifth of ten coding rounds), totalling 384 signals. Fig. 7b shows the pipelining flip-flops used, each iteration alternates between yellow and green.

The output of a secure cryptographic function should be uniformly distributed; the output should resemble a uniform random number generator. The monobit test [19], counts the number of '1's in a data stream. Over a long sequence, the number of '0's and '1's should match, within some statistical bound. We attach three such assertions into the AES circuit, one per encoder, then `AND` these results, driving an off-chip LED. The monobit circuit counts the number of '1's per 128 bit vector, accumulated over 256 cycles (making a stream of 32,768



(a) *Step 1*: Floorplan of the place-and-routed user circuit; each en-/de-coder pair shown in different colours.

(b) *Step 6*: Final floorplan for augmented circuit: inserted logic in white, example signal routing path in cyan.

Fig. 7: Adding 3× 128-bit monobit assertions to the AES (3-pair) benchmark, while maintaining 237MHz. An unusable region in the centre of the FPGA device is also shown.

bits). A range check tests that the number of '1's lies in bounds: for a statistical significance p-value < 0.01, this is $\frac{32768}{2} \pm 466$. In total, the three monobit circuits consume 155 logic slices and 567 LUTs, with a pre-routing timing estimate of 2.44ns.

Using our assertion language, a monobit test looks like:

```
1 user int<N> popcnt( int<2^N> );
2 assertion monobit( int<N> input, int<15> A,
3   int<15> B ) {
4   var count = counter(0, 256);
5   var inp = accum(
6     ((popcnt(input[127:96]) + popcnt(input[95:64]))
7       (popcnt(input[63:32]) + popcnt(input[31:0]))),
8     count==0);
9   (A < inp) ∧ (inp < B); }
```

where line 1 declares a user-defined block to count high bits; lines 2-9 form the assertion, declaring a counter (line 4), accumulating population counts while the counter is non-zero (lines 5-8), testing the range condition (line 9).

Fig. 7b shows the final merged circuit floorplan: assertion circuit logic in white; pipeline signal routing for one signal in cyan. After routing the merged circuit, preserving all existing user nets, static timing analysis by Xilinx tools shows no effect on the critical-path; the circuit still meets timing at 237MHz.

Compared to resynthesising the circuit (with assertions) shows negligible effects (7ps improvement) on critical-path delay between original and instrumented circuits, over five placement seeds. By chance, the best placement in both cases is found with seed value 5; examining other seeds (Table II) shows significant deviations between the two synthesis solutions: for the default seed value of 1, this timing impact exceeds 10%. The runtime improvement for the transparent approach on this circuit is 3.0 times; while the routing runtime has decreased due to it being a less complex circuit (only one clock domain), we must invoke our pipeline-and-route tool five times.

C. Experiment 3 — complex assertion for AES (2-pair): this uses 2 pairs of the AES encoder/decoder circuit, occupies 69%

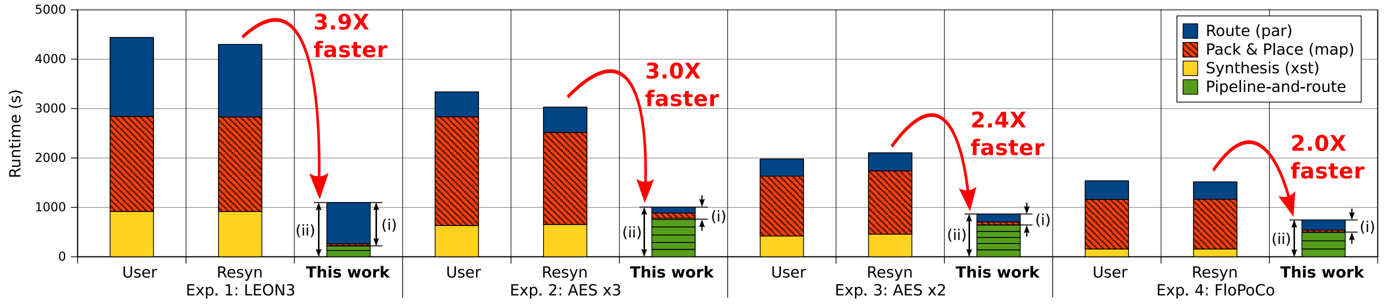


Fig. 6: Runtime comparison between original user circuit compilation (User), resynthesis with new logic (Resyn), and our approach. The runtime for use case (i) — exclusive of step 3, pipeline-and-route — and use case (ii), inclusive, are also shown.

of logic slices and 47% of LUTs, running at 241MHz.

We route two 128-bit buses from each of the two encoder blocks in this benchmark (totalling 512-bits) into the top right region of the device, applying a more complex pattern counter test to each. This divides each 128-bit value into disjoint 4-bit segments, counting the occurrences of each 4-bit pattern. Like the monobit test, over a long stream of bits, each of the 2^4=16 possible patterns should be equally probable. The four pattern counters occupy 1,155 logic slices and 4,262 LUTs.

Using our method does not affect the original critical-path delay (4.15ns). Inserting the same assertion at source level and resynthesising degrades the critical-path delay to 4.32ns (232MHz). The assertion code resembles the monobit test.

C. Experiment 4 — FloPoCo assertion: The final experiment uses our FloPoCo design. With shift-register extraction disabled, the benchmark utilises 65% of all logic slices, 41% of all LUTs, with a critical-path delay of 6.23ns (160MHz). The assertion checks for infinity or NaN conditions at each tap in this pipeline. Each condition is represented in FloPoCo's internal format by one bit going high; for all taps this totals 144 bits.

Rather than just signalling if any assertion fails, we build a priority encoder to transform the 144 bit input into an 8 bit encoded output, to assist a designer in locating the failure.

The FloPoCo assertion can be defined as follows:

```
1  user uint<3> pri( uint<8> ) { latency=... };
2  assertion inRange( uint<34> input[24][8] ){
3    pri(input[0][0][32] @ input[0][1][32] @ ...
4        @ input[0][7][32])
5    @ pri(input[1][0][32] @...@ input[1][7][32])
6    ...
7    @ pri(input[23][0][32] @...@ input[23][7][32])
8    @ pri(input[0][0][33] @...@ input[0][7][33])
9    ...
10   @ pri(input[23][0][33] @...@ input[23][7][33]); }
```

where line 1 declares the priority encoder as a user-defined block, lines 2-10 define the assertion whose inputs are a 24×8 array of 32-bit floating-point numbers, and which concatenates the output of priority encoders whose inputs are bits 32 and 33 of each array element – the NaN and infinity bits of each tap in each parallel filter. Future versions of our assertion language will add loops to ease generation of repetitive assertions.

This assertion circuit is also successfully added into the user circuit without impacting the critical-path delay, while resynthesis with the same placement seed degraded the maximum frequency from 160MHz to less than 100MHz. Over five seeds, the best resynthesis result was 162MHz as shown in Table II.

Clock speed →	Exp. 1: LEON3 75MHz	Exp. 2: AES x3 66MHz	Exp. 2: AES x3 150MHz
User	3.32W	6.00W	11.42W
Resynthesis	3.32W	6.03W	11.57W
This work	3.32W	6.09W	11.68W

Clock speed →	Exp. 3: AES x2 66MHz	Exp. 3: AES x2 200MHz	Exp. 4: FloPoCo 150MHz
User	4.59W	10.36W	5.69W
Resynthesis	4.65W	10.61W	5.73W
This work	4.75W	10.88W	5.72W

TABLE III: Measured power consumption.

E. Power evaluation: Lastly, we investigate the power usage of circuits with and without assertions. We employ the ML605's support for on-chip power measurement (via the Virtex6's System Monitor) – results in Table III show power consumption: for the original user circuit without assertion checking; for assertions added at source level where the entire circuit is resynthesised; for the transparent approach (this paper). All power measurements used ChipScope Analyzer averaged over 128 seconds, once the die temperature had stabilised.

For experiment 1, we boot a Linux image supporting up to 4 cores on the SoC, stressing each core using a gzip instance sourced from `/dev/urandom`. For experiments 2 and 3 based on variants of the self-stimulating AES benchmark, we collect results at two different clock rates. Unfortunately, the high device temperature/current caused by running 'AES x3' at 200MHz triggers the power regulator's shutdown mechanism, so we only show results at 150MHz.

The results show that, unsurprisingly, adding extra assertion circuitry increases power consumption — on average by 2% for resynthesis, and 4% for our techniques. Although resynthesis may be more efficient (smaller area due to denser packing decisions) than the original user circuit, our approach consumes more power due to transporting all assertion inputs, via pipelining registers, into one region to feed the assertion circuit. This incurs multiple hops of extra switching activity not existing in the resynthesis approach, which can distribute the assertion logic close to the signal source without pipelining. For a circuit resynthesised with assertion logic, however, unless additional gating techniques are used this 2% power overhead is permanent, while for our approach it is only temporary — if the assertion logic is no longer needed, the 4% overhead can be recovered by reverting to the original bitstream.

VIII. Conclusion

We propose a language for describing in-circuit hardware assertions in a HDL-agnostic manner, and describe methods to insert latency-oblivious assertion circuitry into a synthesised circuit transparently. Our flow inserts new circuitry *after* the user circuit has been placed-and-routed, using only spare resources; assertions can be added, changed, or removed without affecting the original circuit. To maintain critical-path delay, we aggressively pipeline both the newly inserted circuit and the routing for its inputs. To pipeline signals, we use min-cost flow techniques to efficiently transport signals via pipelining registers, placing and routing them simultaneously.

The key benefits for transparent insertion are: *a*) only spare resources are needed, even on large, complex designs; *b*) the critical-path delay is unaffected, *c*) it is 2–3.9-fold faster than resynthesis. Our approach incurs a small, temporary, power overhead: extra switching from pipelining new circuit inputs.

We further extend our technique to allow in-circuit exceptions; by relaxing strict transparency, some circuit errors can be fixed without rerunning expensive place-and-route.

Currently, our transparent insertion flow is encumbered by overly-broad constraints, owing to using the Xilinx toolflow for an unsupported application. When building inserted circuit (step 4 of our flow) we can only mark logic resources as occupied at slice granularity — even if only one of four slice LUTs is occupied, we cannot use the rest of the slice; furthermore, we cannot mark occupied routing resources in the same manner.

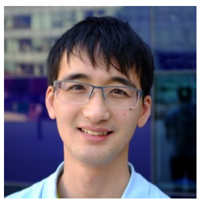
Furthermore, we must use constraints to force inserted circuits to be placed near the pipelined signals, to minimise routing congestion between user and inserted circuits, given that the current flow compiles the inserted circuit without knowledge of leftover routing. These limitations may be lifted by building toolflows to create and insert transparent circuits, e.g. modifying the VTR-to-Bitstream project [20].

In the long term, we would like to consider enhancements to FPGA architectures and CAD toolflows to further improve the effectiveness of inserted assertions and exceptions.

Acknowledgements: This work is supported in part by the European Union Seventh Framework Programme under grant agreements 257906, 287804 and 318521, by the UK EPSRC, by the Maxeler University Programme, by the HiPEAC NoE, by Altera, and by Xilinx.

References

[1] K. Murray, S. Whitty *et al.*, “Titan: Enabling Large and Complex Benchmarks in Academic CAD,” in *2013 Int’l Conf. on Field Programmable Logic and Applications (FPL)*, Sept 2013, pp. 1–8.
[2] R. Y. Rubin and A. M. DeHon, “Timing-driven Pathfinder Pathology and Remediation: Quantifying and Reducing Delay Noise in VPR-pathfinder,” in *2011 Int’l Symp. on Field Programmable Gate Arrays (FPGA)*, 2011, pp. 173–176.
[3] E. Hung, A.-S. Jamal, and S. Wilton, “Maximum Flow Algorithms for Maximum Observability during FPGA Debug,” in *2013 Int’l Conf. on Field-Programmable Technology (FPT)*, Dec 2013, pp. 20–27.
[4] J. Curreri, G. Stitt, and A. D. George, “High-level Synthesis of In-Circuit Assertions for Verification, Debugging, and Timing Analysis,” *Int. J. Reconfig. Comput.*, vol. 2011, pp. 1:1–1:17, Jan. 2011.
[5] E. Hung, T. Todman, and W. Luk, “Transparent Insertion of Latency-Oblivious Logic onto FPGAs,” in *2014 Int’l Conf. on Field-Programmable Logic and Applications (FPL)*, September 2014, pp. 1–8.
[6] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, “Theory of latency-insensitive design,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.
[7] T. Todman, S. Stilkerich, and W. Luk, “Using Statistical Assertions to Guide Self-Adaptive Systems,” in *Proc. of the Workshop on Self-Awareness in Reconfigurable Systems (SRCS)*, Sept 2013, pp. 28–32.
[8] D. Bustan, D. Korchemny *et al.*, “SystemVerilog Assertions: Past, Present, and Future SVA Standardization Experience,” *Design Test of Computers, IEEE*, vol. 29, no. 2, pp. 23–31, April 2012.
[9] IEEE, “IEEE Standard for Property Specification Language (PSL),” *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, April 2010.
[10] S. Das, R. Mohanty *et al.*, “Synthesis of System Verilog Assertions,” in *Design, Automation and Test in Europe, 2006. DATE ’06. Proc.*, vol. 2, March 2006, pp. 1–6.
[11] J. Cong and Y. Ding, “FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.
[12] G. Lemieux, P. Leventis, and D. Lewis, “Generating highly-routable sparse crossbars for PLDs,” in *2000 Int’l Symp. on Field Programmable Gate Arrays (FPGA)*, 2000, pp. 155–164.
[13] F. Eslami and S. J. E. Wilton, “Incremental distributed trigger insertion for efficient FPGA debug,” in *2014 Int’l Conf. on Field-Programmable Logic and Applications (FPL)*, September 2014.
[14] N. Steiner, A. Wood *et al.*, “Torc: Towards an Open-Source Tool Flow,” in *2011 Int’l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb. 2011, pp. 41–44.
[15] B. Dezs, A. Jüttner, and P. Kovács, “LEMON - an Open Source C++ Graph Template Library,” *Electron. Notes Theor. Comput. Sci.*, vol. 264, no. 5, pp. 23–45, July 2011.
[16] Aeroflex Gaisler, “GRLIB IP Core User’s Manual,” http://www.gaisler.com/products/grlib/grip.pdf, January 2013.
[17] Altera, “Advanced Synthesis Cookbook,” http://www.altera.co.uk/literature/manual/stx_cookbook.pdf, July 2011.
[18] F. de Dinechin and B. Pasca, “Designing Custom Arithmetic Data Paths with FloPoCo,” *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.
[19] L. E. Bassham III, A. L. Rukhin *et al.*, “SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications,” National Institute of Standards & Technology, United States, Tech. Rep., 2010.
[20] E. Hung, F. Eslami, and S. Wilton, “Escaping the Academic Sandbox: Realizing VPR Circuits on Xilinx Devices,” in *2013 Int’l Symp. on Field-Programmable Custom Computing Machines (FCCM)*, April 2013, pp. 45–52.



Eddie Hung is currently the technical lead at Invionics, a startup offering a platform for building custom EDA tools. He received his M.Eng. degree from the University of Bristol, Bristol, UK, in 2008, and his Ph.D. from the University of British Columbia, Vancouver, BC, Canada in 2013. The research described in this manuscript was completed when he was a post-doc at Imperial College London, London, UK during 2014-15.

Tim Todman received his B.Sc. degree from the University of North London and M.Sc. and Ph.D. degrees from Imperial College London, London, U.K. He is a research associate in the Department of Computing, Imperial College London. His research interests include high-level synthesis and runtime verification of reconfigurable hardware designs.

Wayne Luk (F09) received the M.A., M.Sc., and D.Phil. degrees in engineering and computing science from the University of Oxford, Oxford, U.K. He was a Visiting Professor with Stanford University, Stanford, CA, USA. He is currently a Professor of Computer Engineering with Imperial College London, London, U.K. His current research interests include reconfigurable computing, field programmable technology, and design automation.