Transparent Insertion of Latency-Oblivious Logic onto FPGAs

Eddie Hung, Tim Todman, Wayne Luk
{e.hung, timothy.todman, w.luk}@ic.ac.uk

Department of Computing
Imperial College London, UK

FPL14 :: 3 Sept '14
What this talk is all about...

- Automatically add functionality to an FPGA circuit without modifying it
  - Post place-and-route, using only *spare resources*
What this talk is all about...

- Automatically add functionality to an FPGA circuit without modifying it
  - Post place-and-route, using only *spare resources*
- Novelty: decouple input signals to new circuitry by pipelining
  - Circuit must be oblivious to this latency
- e.g.: self-monitoring circuitry
  - Demonstrate no timing impact on large >200MHz designs evaluated on Xilinx technology
Introduction

- FPGAs are a general-purpose silicon technology
  - Flexibility from provisioning configurable resources, such as LUTs, FFs, switched routing, etc.
  - Not all resources are/can be used

- Mapping designs onto FPGAs requires invoking CAD tools
  - Changing designs often require a recompile
Introduction

• FPGA CAD tools are imperfect
  – Tool runtimes are lengthy – recompiling undesirable
  – Complexity necessitates use of heuristic algorithms
    • Even small changes can drastically affect solution quality

• This work: how can you augment your FPGA design without affecting its Fmax?
Introduction

• We focus on inserting *latency-oblivious* logic
  - Circuitry where you can add any number of pipeline stages to without affecting its correctness
  - Examples:
    • Trace-buffers for debug
    • Self-monitoring circuitry (e.g. system watchdog)
  - **Key advantage:** conceding latency provides an additional level of CAD flexibility
Contributions

1. Reclaiming spare, unused, FPGA resources to transparently insert new logic

2. Use of network flow techniques to simultaneously pipeline-and-route signals, without affecting timing

3. Experimentally quantifying overhead of inserting self-monitoring circuit on Xilinx FPGAs
   - Finding no impact on Fmax, <3% power overhead, on designs over 200MHz
Challenge

• Spare logic resources are typically available far away from any interesting signals

(Unconstrained floorplan of a benchmark, blue indicates logic cluster partially used)
Challenge

- Spare logic resources are typically available far away from any *interesting* signals
- Whilst possible to drop circuitry there, routing delays would create a huge impact
Solution: Pipeline-and-Route

- To mitigate any delay impact, we pipeline signals

- Unique opportunity:
  
  *any signal can use any spare register for pipelining as they are all equal*
Solution: Pipeline-and-Route

• To mitigate any delay impact, we pipeline signals

• Unique opportunity:

  *any signal can use any spare register for pipelining as they are all equal*

  - This fits into the **network flow** paradigm!
  - Instead of place-then-routing pipelining registers, we do place-*while*-routing
Solution: Pipeline-and-Route

- Network flow studies how to transport a commodity from source $s$ to sink $t$:

![Spare Routing Resource Graph]

Signal sources

Spare registers
Solution: Pipeline-and-Route

- Network flow studies how to transport a commodity from source $s$ to sink $t$:
Solution: Pipeline-and-Route

- Network flow studies how to transport a commodity from source $s$ to sink $t$:

  Flow algorithm will automatically choose best placement!

  $\Rightarrow$ Only place while routing
Solution: Pipeline-and-Route

- We iteratively migrate the signals of interest towards an 'anchor' point – spare resources.
Solution: Pipeline-and-Route

- We iteratively migrate the signals of interest towards an 'anchor' point – spare resources

(a) First hop
Solution: Pipeline-and-Route

- We iteratively migrate the signals of interest towards an 'anchor' point – spare resources

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

1. Compile user-circuit
2. Identify underutilised region
3. Iteratively pipeline-and-route, using only **spare** resources
4. Compile new logic on spare resources
5. Merge user-circuit with new logic
6. Finalise routing on spare resources

(XDL)
Visualisation
Visualisation
Visualisation
Visualisation
Visualisation
Evaluation

• Experiments on Xilinx ISE 13.3 targeting Virtex6 ML605

• Example: AES encoder/decoder 3x chain
  – Baseline:
    92% logic slices, 71% LUTs, 10% regs, >200MHz
  – Insert simple monitoring circuit to verify each encoder is secure, by checking if data outputs are statistically random
Results: AES x3

• Compared with recompiling, our work:
  – Inserted monitoring circuit with 3X faster CAD runtime
  – Preserved all aspects of the user-circuit, including its critical-path delay
Results: AES x3

• Compared with recompiling, our work:
  – Inserted monitoring circuit with 3X faster CAD runtime
  – Preserved all aspects of the user-circuit, including its critical-path delay
  – Reclaimed new FPGA resources (which were lying unused anyway)
  – Incurred a small, measured, power overhead:
    • Original: 11.42W
    • Recompile: 11.57W
    • This work: 11.68W (<1% extra over recompile)
Results

- Similar findings for our other benchmarks:
  - No delay, <3% power penalty, 4X faster CAD
  - LEON3 System-on-Chip
    - Program counter monitor
  - AES 2x chain
    - Complex variant of randomness monitor
  - FloPoCo floating point datapath
    - Floating point exception monitor

- More details, more analysis, in our paper!
Conclusion

- A method to automate inserting latency-oblivious logic into existing circuits, transparently
  - Enabled by use of network flow techniques to pipeline-and-route signals of interest

- Key benefits:
  - Only spare resources are needed
  - Critical-path delay is unaffected
  - 2 to 3.9X faster CAD than full recompilation

- Future work: engineer a toolflow tailored towards inserting transparent circuits
Backup Slides
Solution: Pipeline-and-Route

- Traditionally: place-then-route
Solution: Pipeline-and-Route

- Traditionally: \textbf{place}-then-route
Solution: Pipeline-and-Route

- Traditionally: place-then-route
Solution: Pipeline-and-Route

• Traditionally: place-then-route

![Diagram showing traditional place-then-route methodology]

• Our work: **place-while-route**, considers all spare registers for all pipelining signals

![Diagram showing proposed place-while-route methodology]
Solution: Pipeline-and-Route

- Traditionally: place-then-route

- Our work: **place-while-route**, considers all spare registers for all pipelining signals
Solution: Pipeline-and-Route

• Traditionally: place-then-route

• Our work: place-while-route, considers all spare registers for all pipelining signals
CAD Runtime

![Bar chart comparing runtime for User, Resyn, and This work in experiments 3 and 4.](chart.png)

- **Route (par)**
- **Pack & Place (map)**
- **Synthesis (xst)**
- **Pipeline-and-route**

**Exp. 3: AES x2**
- **User**
- **Resyn**
- **This work**

- 2.4X faster

**Exp. 4: FloPoCo**
- **User**
- **Resyn**
- **This work**

- 2.0X faster
CAD Runtime

3.9X faster

3.0X faster

Runtime (s)

User

Resyn

Exp. 1: LEON3

This work

User

Resyn

Exp. 2: AES x3

This work
AES x3 benchmark
AES x2 randomness monitor

128b input

32b  32b  32b  32b

PopCnt  PopCnt  PopCnt  PopCnt

5b + 5b  5b + 5b

6b + 6b

7b

rst +

15b

A < input < B

Modulo 256 counter

1b

Input registers (1 stage)

Bit population counters (2 stages)

Pipelined adder tree (3 stages)

15 bit accumulator (1 stage)

Range check

Output register, with enable (1 stage)
Results: LEON3 & AES x3

<table>
<thead>
<tr>
<th></th>
<th>Exp. 1: LEON3 SoC</th>
<th>Exp. 2: AES (3 pair)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This work</td>
<td>Resynthesis</td>
</tr>
<tr>
<td><strong>User circuit:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slice utilization</td>
<td>30,698 (81%)</td>
<td>34,880 (92%)</td>
</tr>
<tr>
<td>LUT utilization</td>
<td>82,830 (54%)</td>
<td>108,132 (71%)</td>
</tr>
<tr>
<td>Register utilization</td>
<td>60,725 (20%)</td>
<td>32,022 (10%)</td>
</tr>
<tr>
<td>Critical-path delay</td>
<td>13.324ns</td>
<td>4.213ns</td>
</tr>
<tr>
<td><strong>Pipe-and-routed ckt:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals routed</td>
<td>240</td>
<td>384</td>
</tr>
<tr>
<td>Slice utilization</td>
<td>30,720 (+22)</td>
<td>34,985 (+105)</td>
</tr>
<tr>
<td>LUT utilization</td>
<td>82,925 (+95)</td>
<td>108,264 (+132)</td>
</tr>
<tr>
<td>Register utilization</td>
<td>61,205 (+480)</td>
<td>33,942 (+1,920)</td>
</tr>
<tr>
<td>Critical-path delay</td>
<td>13.324ns</td>
<td>4.213ns</td>
</tr>
<tr>
<td>Pipeline latency</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td><strong>Monitoring circuit:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slice utilization</td>
<td>30,770 (+50)</td>
<td>35,140 (+155)</td>
</tr>
<tr>
<td>LUT utilization</td>
<td>83,078 (+153)</td>
<td>108,831 (+567)</td>
</tr>
<tr>
<td>Register utilization</td>
<td>61,454 (+249)</td>
<td>34,636 (+694)</td>
</tr>
<tr>
<td>Critical-path estimate</td>
<td>3.729ns</td>
<td>2.436ns</td>
</tr>
<tr>
<td>Monitor latency</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Final circuit:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Critical-path delay</td>
<td>13.324ns</td>
<td>13.327ns</td>
</tr>
<tr>
<td></td>
<td>4.213ns</td>
<td>4.205ns</td>
</tr>
</tbody>
</table>
## Results: AES x2 & FloPoCo

<table>
<thead>
<tr>
<th>User circuit:</th>
<th>Exp. 3: AES (2 pair)</th>
<th>Exp. 4: FloPoCo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This work</td>
<td>Resynthesis</td>
</tr>
<tr>
<td>Slice utilization</td>
<td>26,362 (69%)</td>
<td>24,650 (65%)</td>
</tr>
<tr>
<td>LUT utilization</td>
<td>71,976 (47%)</td>
<td>61,967 (41%)</td>
</tr>
<tr>
<td>Register utilization</td>
<td>21,391 (7%)</td>
<td>97,968 (32%)</td>
</tr>
<tr>
<td>Critical-path delay</td>
<td>4.153ns</td>
<td>6.232 ns</td>
</tr>
<tr>
<td><strong>Pipe-and-routed ckt:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals routed</td>
<td>512</td>
<td>144</td>
</tr>
<tr>
<td>Slice utilization</td>
<td>26,890 (+528)</td>
<td>24,790 (+140)</td>
</tr>
<tr>
<td>LUT utilization</td>
<td>72,216 (+240)</td>
<td>61,996 (+29)</td>
</tr>
<tr>
<td>Register utilization</td>
<td>23,951 (+2,560)</td>
<td>98,400 (+432)</td>
</tr>
<tr>
<td>Critical-path delay</td>
<td>4.153ns</td>
<td>6.232ns</td>
</tr>
<tr>
<td>Pipeline latency</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td><strong>Monitoring circuit:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slice utilization</td>
<td>28,045 (+1155)</td>
<td>25,807</td>
</tr>
<tr>
<td>LUT utilization</td>
<td>76,478 (+4262)</td>
<td>75,996</td>
</tr>
<tr>
<td>Register utilization</td>
<td>28,385 (+4434)</td>
<td>27,765</td>
</tr>
<tr>
<td>Critical-path estimate</td>
<td>2.758ns</td>
<td>-</td>
</tr>
<tr>
<td>Monitor latency</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td><strong>Final circuit:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Critical-path delay</td>
<td>4.153ns</td>
<td>4.318ns</td>
</tr>
</tbody>
</table>


# Measured Power Consumption

<table>
<thead>
<tr>
<th>Clock speed →</th>
<th>Exp. 1: LEON3 75MHz</th>
<th>Exp. 2: AES x3 66MHz</th>
<th>Exp. 2: AES x3 150MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>3.32W</td>
<td>6.00W</td>
<td>11.42W</td>
</tr>
<tr>
<td>Resynthesis</td>
<td>3.32W</td>
<td>6.03W</td>
<td>11.57W</td>
</tr>
<tr>
<td><strong>This work</strong></td>
<td>3.32W</td>
<td>6.09W</td>
<td>11.68W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clock speed →</th>
<th>Exp. 3: AES x2 66MHz</th>
<th>Exp. 3: AES x2 200MHz</th>
<th>Exp. 4: FloPoCo 150MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>4.59W</td>
<td>10.36W</td>
<td>5.69W</td>
</tr>
<tr>
<td>Resynthesis</td>
<td>4.65W</td>
<td>10.61W</td>
<td>5.73W</td>
</tr>
<tr>
<td><strong>This work</strong></td>
<td>4.75W</td>
<td>10.88W</td>
<td>5.72W</td>
</tr>
</tbody>
</table>

**TABLE III: Measured power consumption.**
Pipeline-and-Route tool

From Step 1
- Circuit Netlist (XDL)
- Signals to route (regex)
- Clock Signal
- Anchor point (X,Y,radius)

Re-entrant capability for iterative application

From Step 2

Pipeline-and-Route tool

To Step 5
- Modified Circuit Netlist (XDL)
- Insertion Template (Verilog & UCF)

To Step 4
LEON3 visualisation

(81% logic slice utilisation)
AES x2 visualisation

(69% logic slice utilisation)
Network flow

• Express trace-buffer routing problem as graph:

Interesting Signals  Spare FPGA wires  Spare registers
Network flow

- Transform routing graph into flow network:
  - All edges have capacity of one

“Super source” — “Node splitting” — “Super sink”
Network flow

- Transform routing graph into flow network:
  - All edges have capacity of one, congestion cost
  - Goal: minimise global cost