RapidWright¹:
Enabling Custom Crafted Implementations for FPGAs

Chris Lavin & Alireza Kaviani
FCCM - May 1, 2018

Wright¹ = maker or builder
FPGA Implementation Tools Lead a Challenging Life...

Oh, and make it easy to use for software programmers 😊

Highest QoR Possible!!

10X faster compile time!!

For Every Imaginable Design!!

Highest utilization possible!!
What is RapidWright?

> Companion framework for Vivado
  - Communicates through DCPs
  - Fast, light-weight, open source
  - Java, Python support

> Enables targeted solutions
  - Reuse & relocate pre-implentated modules
  - Systematic shells & overlays
  - Generate on-the-fly implementations

> Academic ecosystem
  - Algorithm validation
  - Rapid prototyping of CAD concepts
RapidWright Framework Overview

- **Vivado**
  - .dcp
  - Labs Vivado
  - .xdd

- **RapidWright**
  - EDIF Reader/Writer
  - XDEF Reader/Writer
  - CHECKPOINT READER/WRITER
  - Device Files Creator
  - XDD Parser

- **Logical Netlist**
  - Physical Netlist
  - DESIGN MODEL

- **APPLICATIONS**
  - Device Netlist
  - .dat
  - ARCH MODEL
  - PART GENERATION
Modular Pre-Implemented Methodology

1. Restructure Design
2. Packing & Placement Planning
3. Stitch, Place & Route Implementation

USER COMPLETED
Design

USER ASSISTED

TOOL AUTOMATED
Vivado OOC Flow
IP Cache*
RapidWright
Creating Pre-implemented Modules (Vivado OOC Flow)

OOC Synthesis

VIVADO

PBlock Generator

(...or user supplied)

OOC Place & Route

VIVADO

Block Generator

IP Cache++
Step 3: RapidWright Pre-implemented Module Flow

IPI Design

Vivado OOC Flow

IP Cache*

rw.dcp

Route Design

Routed .dcp

Impl Guide File

IPI Design Parser

Block Stitcher

Block Placer

*IP Cache Augmented w/RapidWright
### Design Results

<table>
<thead>
<tr>
<th>Design</th>
<th>Target Device</th>
<th>Baseline (initial design)</th>
<th>Vivado (restructured design)</th>
<th>Δ</th>
<th>Pre-implemented Flow</th>
<th>Δ</th>
<th>Total Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seismic</td>
<td>KU040</td>
<td>270MHz</td>
<td>354MHz</td>
<td>31%</td>
<td>390MHz</td>
<td>10%</td>
<td>41%</td>
</tr>
<tr>
<td>FMA</td>
<td>KU115</td>
<td>270MHz</td>
<td>273MHz</td>
<td>1%</td>
<td>417MHz</td>
<td>53%</td>
<td>54%</td>
</tr>
<tr>
<td>GEMM</td>
<td>KU115*</td>
<td>391MHz</td>
<td>437MHz</td>
<td>10%</td>
<td>462MHz</td>
<td>6%</td>
<td>16%</td>
</tr>
<tr>
<td>Heart of Gold</td>
<td>ZU9EG*</td>
<td>368MHz</td>
<td>569MHz</td>
<td>55%</td>
<td>541MHz</td>
<td>-5%</td>
<td>50%</td>
</tr>
</tbody>
</table>

*Constrained area of the device

<table>
<thead>
<tr>
<th>Design</th>
<th>LUT Utilization</th>
<th>FF Utilization</th>
<th>DSP Utilization</th>
<th>BRAM Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seismic</td>
<td>93% (226k)</td>
<td>5% (26k)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FMA</td>
<td>25% (166k)</td>
<td>50% (656k)</td>
<td>97% (5360)</td>
<td>6% (130)</td>
</tr>
<tr>
<td>GEMM</td>
<td>19% (64k)</td>
<td>20% (134k)</td>
<td>87% (2400)</td>
<td>-</td>
</tr>
<tr>
<td>Heart of Gold</td>
<td>46% (30k)</td>
<td>29% (38k)</td>
<td>42% (272)</td>
<td>96% (208)</td>
</tr>
</tbody>
</table>

1Speed Grade: -2
2. FMA Design

- **GOAL**
  - Highest compute (TeraOp/s) possible
  - 16-bit fused multiply accumulate

- **KU115 Implementation**
  - 1340 kernel instances
    - 4x10 CLEs, 1x4 DSPs
  - 97% DSP utilization
  - 4.4 TeraOp/s

- “Fabric discontinuites”
  - SLR boundary
  - IO Columns
  - Depopulated CLEs (SLR crossing)

<table>
<thead>
<tr>
<th>RUN</th>
<th>$F_{\text{MAX}}$ (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (initial design)</td>
<td>270</td>
</tr>
<tr>
<td>Vivado (restructured design)</td>
<td>273 (+1%)</td>
</tr>
<tr>
<td>Pre-implemented Flow</td>
<td>417 (+53%)</td>
</tr>
</tbody>
</table>

FMA KERNEL
Re-locatability & Multiple Implementations

Impl #0
Impl #1
Impl #2
Impl #3
Impl #4
Impl #5

IO

IO
Crossing IOs: AXI Stream Register Slices

- Strategically placed register slices to cross long distances
  - cross chip IO columns
- Latency insertion/connectivity is easily automated
Debug Productivity

Leverage unused resources to place blocks and route probes.
RapidWright vs. Vivado for Debug Instrumentation Speedup

- **Baseline**
- **RapidWright Debug**

The graph shows the runtime (in minutes) for various designs with different CLB (Configurable Logic Block) percentages. The y-axis represents the runtime, and the x-axis lists different designs with their respective CLB percentages:

- **dsp1 (9% CLBs)**: Baseline runtime of 35 minutes, RapidWright Debug runtime of 0.0 minutes (35x speedup)
- **10g (10% CLBs)**: Baseline runtime of 24 minutes, RapidWright Debug runtime of 0.0 minutes (24x speedup)
- ** dsp2 (20% CLBs)**: Baseline runtime of 12 minutes, RapidWright Debug runtime of 0.0 minutes (12x speedup)
- **sparc (31% CLBs)**: Baseline runtime of 97 minutes, RapidWright Debug runtime of 0.0 minutes (97x speedup)
- **21ch (70% CLBs)**: Baseline runtime of 33 minutes, RapidWright Debug runtime of 0.0 minutes (33x speedup)
Fully Placed and Routed Designs in Seconds

<table>
<thead>
<tr>
<th>Design</th>
<th>RapidWright Flow</th>
<th>Vivado</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>microblazeDesign</td>
<td>12.5 seconds</td>
<td>232 seconds</td>
<td>19x</td>
</tr>
<tr>
<td>fpFIRFilter</td>
<td>18.6 seconds</td>
<td>183 seconds</td>
<td>10x</td>
</tr>
</tbody>
</table>
On-the-fly, Pre-implemented Module Generators (Demo)

- Build modules on-demand
  - Placed and routed *in seconds*
  - Reusable and compose-able
  - Target spec performance

- Parameterizable Generators
  - Adder
  - Subtractor
  - Multiplier

- Polynomial Solution Generator
  - Runs at spec 775MHz (UltraScale+, SG2)
  - Constructed on-the-fly in seconds
  - Still in development

\[ x^2 + 3x - 5 \]
Goal: Achieve Spec Performance
- 775MHz on UltraScale+, SG2
- Minimize overhead of overlays/shells

LinkBlaze [1]: Data movement soft NoC
- 128 bit, bi-directional
- Modular design
  • Pre-implemented modules
  • Captures high performance implementation

Challenge: Crossing SLR
- Solved using two techniques in RapidWright
  • Custom clocking of Leaf clock buffer and delay tuning
  • Custom clock root routing per SLR crossing

[1] LinkBlaze: Efficient global data movement for FPGAs (ReConFig 2017)
SLR Crossing Solution – Clocking Techniques

- Leaf Clock Buffers (LCBs)
  - Custom route RX/TX to same LCB
  - Tune LCB delay to avoid hold issues

- Inter-SLR Compensation
  - 15% tax of clk delay between root and RX flop
  - Minimize by custom creating clock roots per SLR crossing

### Vivado 2017.3

<table>
<thead>
<tr>
<th>LCB</th>
<th>LCB &amp; Custom Clk Route</th>
</tr>
</thead>
<tbody>
<tr>
<td>549MHz</td>
<td>595MHz</td>
</tr>
</tbody>
</table>

© Copyright 2018 Xilinx
RapidWright SLR Crossing DCP Generator (Demo)

SLR crossing module from scratch
- Parameterizable
- Closes timing at 760MHz
  - Clk Period: 1.313ns
- Routed clock, placed and routed
- Runs in seconds

---
SLR Crossing DCP Generator
---

This RapidWright program creates a placed and routed DCP that can be imported into UltraScale+ designs to aid in high speed SLR crossings. See RapidWright documentation for more information.

Option                                Description
-----                                  -------
-? -h                                  Print Help
-a [String: Clk input net name]        (default: clk_in)
-b [String: Clock BUFGCE site name]    (default: BUFGCE_X0Y218)
-c [String: Clk net name]              (default: clk)
-d [String: Design Name]               (default: slr_crosser)
-i [String: Input bus name prefix]     (default: input)
-l [String: Comma separated list of Laguna sites for each SLR crossing]
-n [String: North bus name suffix]     (default: _north)
-o [String: Output DCP File Name]      (default: slr_crosser.dcp)
-p [String: UltraScale+ Part Name]     (default: xcvu9p-flgc2104-2-i)
-q [String: Output bus name prefix]    (default: output)
-r [String: INT clk Laguna RX flops]   (default: GCLK_B_0_1)
-s [String: South bus name suffix]     (default: _south)
-t [String: INT clk Laguna TX flops]   (default: GCLK_B_0_0)
-u [String: Clk output net name]       (default: clk_out)
-v [Boolean: Print verbose output]     (default: true)
-w [Integer: SLR crossing bus width]   (default: 512)
-x [Double: Clk period constraint (ns)] (default: 1.538)
-y [String: BUFGCE cell instance name] (default: BUFGCE_inst)
-z [Boolean: Use common centroid]       (default: false)
Vision: Pre-implemented Modules

Parameterizable Circuit Generators

Algorithmic Engines (SAT Solvers, ILP, ...)

Vivado-optimized OOC Solutions

PROBLEM SIZE

COMPLEXITY / DIFFICULTY
Takeaways

RapidWright enables customized solutions
- Relocate & replicate pre-implemented modules
- On-the-fly circuit generators
- Leverage algorithmic engines (SAT Solvers, ILP, …)

Modular pre-implemented Methodology
- Up to 50% performance improvement
- ~10X productivity gains
- Near-spec performance (94% of spec)

www.rapidwright.io
- Open source -- try it out today
- Documentation, tutorials, source code, and demos