18-643 Recitation 6: PR in Real-Life

Shashank Obla
Department of ECE
Carnegie Mellon University
Housekeeping

- Your goal today: experience PR tangibly, understand today’s constraints and what goes into using it
- Notices
  - 1 week down already on the Project!
  - Future recitation slots will be OH
- This is not a lecture! You need to be interactive, lots of discussions, questions and answers
- Readings:
  - Xilinx Dynamic Function Exchange
  - Intel Partial Reconfiguration
Partial Reconfiguration (PR)

- Some parts of fabric retain their configured “personality” while other parts are reconfigured
  - e.g., keep the external bus interface from babbling while the functionality behind is changed
- The alive part can even control the reconfig.
  - e.g., load the bitstream through the bus
- Essential to FPGA as a flexible, sharable computing device
- What does PR let us do which field programmability couldn’t?
This is where we’re at technologically...

FPGAs
Field Programmability

Months

FPGAs
Inter/Intra-Task Time-Multiplexing

100ms ~ sec

FPGAs
Role and Shell use-case

Today we learn how to use PR

Ongoing Research

Years

ASIC
Once in a lifetime

Hr/Days

μs

Processor
Context Switching
PR Conceptually

- Module `top()` instantiate submodules `foo(A)` and `bar(B)` with interface `A` and `B` respectively
  - `foo(A)` and `bar(B)` are “blackboxes”, i.e., interface only, no internals
  - `m1(~) m5()` have matching interfaces, `A` or `B`
Static and Reconfigurable Partitions

The static region including the PR region interfaces are sacred
Bitstream defines the chip

- After power up, SRAM FPGA loads bitstream from somewhere before becoming the “chip”
  - a bonus “feature” for sensitive devices that need to forget what it does
- Many built-in loading options
- Non-trivial amount of time; must control reset timing and sequence with the rest of the system
- Reverse-engineering concerns ameliorated by
  - encryption
  - proprietary knowledge
At Run Time

• Power up with full-design bitstream
• Partial bitstreams in DRAM or flash memory
• Configuration API driven by SoC (e.g., ARM) or fabric
  – PCAP interface a function call from ARM (easy) vs ICAP using DFX Controller IP on fabric (more control)
Concrete Syntax (Xilinx’s approach)

```verilog
module top();
  . . .
  foo instance.foo (a1, a2, ...);
  bar instance.bar (b1, b2, ...);
  . . .
endmodule

module foo(
  input a1, a2, ...
  output ax, ...);
  // nothing here
endmodule

module m1(
  input a1, a2, ...
  output ax, ...
);
  . . . RTL body . . .
endmodule

module m2(
  input a1, a2, ...
  output ax, ...
);
  . . . RTL body . . .
endmodule

module m3(
  input a1, a2, ...
  output ax, ...
);
  . . . RTL body . . .
endmodule
```

Concrete Syntax (Xilinx’s approach)
Implementation Flow

1. Enable DFX in the Vivado Project (no turning back)
2. Designate `instance_foo` and `instance_bar` as *Reconfigurable Partitions* (RPs) – restrictions limit possibilities to non-parametrized RTL top levels
3. Run Implementation and perform Floorplanning
   - draw bounding boxes for RPs
     reserve enough resource for largest variant
   - place partition interface pins (optional)
4. Add Reconfig. Modules (RM) using DFX Wizard
5. Create Configurations: Static Logic + 1 RM per RP
   - Default config should use largest RM variants
Implementation Flow (continued)

6. Create Configuration Runs: Setup hierarchy of runs

   Synth → Impl(config1) → Impl(config3)
   → Impl(config2)

7. Execute all configuration runs till bitstream gen.

End Result

- 1 full-design bitstream per config (including RMs)
- partial design bitstreams individually for m1~m5 in each configuration (could include duplicates)
Prior Flows (Intel still looks like this)

After defining Reconfiguration Partitions:
5. Place-and-route full design with $m_2$ and $m_4$ in RPs
   - extract partial designs of just $m_2$ and $m_4$ RPs
   - extract static partition (full design with blank RPs)
(Re-start from static partition and locked floorplan)
6. Bind `instance_foo` and `instance_bar` to remaining variants (e.g., $m_1$ and $m_5$)
7. Place and route new full design
8. Extract partial designs of just $m_1$ and $m_5$ RPs . . .
   repeat for $m_3$ . . . and so on
What this looks like . . .

PR Region

ARM Core

Static Region
Greybox/Blackbox Flow

• Block-based design need not be limited to PR
  – Think about IPs/shell of Placed and Routed Logic

• Design Partitions can be defined as a greybox
  – Empty box with only interfaces and no logic
  – Tie-off LUTs to seal interfaces, hence grey
  – P&R and optimize surrounding design on fabric

• Could also flip – reuse the partition design

• Use Fast Preserve to only recompile interfaces to get more flexibility while saving compile time

• Reference: [Intel FPGA Block-Based Design](https://www.intel.com/content/www/us/en/fpga/products/design-development/intel-fpga-block-based-design.html)
We saw the basic flow...

DESIGN CONSIDERATIONS FOR PR
Configurable Routing
(1980s Xilinx simplified)
Static – PR Region Interfacing

static partition: top

Handshakes and what not!
Decouple/Freeze Interface

Static partition: top
Advanced Runtime Control

Unbounded time between stop_req and stop_ack

stop_req is deasserted when stop_ack is asserted

start_req is asserted when freeze is deasserted

region_reset is fully user controlled

Unbounded time between start_req and start_ack

start_req is deasserted when start_ack is asserted

Partial Reconfiguration

Source: Intel FPGA PR Guide
Floorplanning and Routing Region

• Recall Rent’s Rule... do you see a problem?

• Ability to increase routing region to reduce congestion

Rent’s Rule

- $T \propto g^p$
  - $T$ = number of inputs and outputs
  - $g$ = number of internal components
  - $p$ typically between 0.5 (regular) and 0.8 (random)
- In a square, perimeter = 4-area$^{0.5}$
  - unless regular, I/O signals grow faster than available routes exiting a design area
- Need hierarchy of progressively longer additional routing resources
  
  long routes also reduce delay when going far

Source: Xilinx DFX Guide
Floorplanning and Timing Closure

Combinational path

Static partition

A

B

m1

Reconfig. partition

Static partition

m1

Reconfig. partition

m2

Reconfig. partition

m4
Floorplanning: Resource Fragmentation

A Reconfigurable Design can only use resources within its region wasting resources which could have otherwise been used...
Nested/Hierarchical PR

- Floorplanning $\Rightarrow$ area and interface constraints
  - But only need to freeze interfaces, free to do whatever inside why not PR within?
  - Fabric is open to anything – just reconfiguration
- Can mix and match number of slots in different parent versions
- Follow the same flow and design considerations throughout

*Figure 9: Two Configurations of a Nested DFX design*

Source: Xilinx DFX Guide
Debugging PR Designs

Source: Intel FPGA PR Guide
Midway Thoughts

- PR is real and not a mere academic venture!
- But yes, the flow is not what it could be and is built as an afterthought with the given resources
  - The fabric has immense flexibility still but sometimes limited by what the tools lets you do
- PR fundamentally changes the way you design
  - Think of every PR region as a smaller FPGA and you’ll the same constraints but in a smaller area
- **Optional:** A thought experiment – find ways to use PR to improve your project – report it!

*We’re not done yet!*
Back to everyone's favorite Matrix Multiplication!

PR DEMO/EXAMPLE
Motivation (for using PR)

- Consider MMM – need to support floating point as well as integer (could imagine other types)
  - Slack: not all at the same time though

- Options:
  - ASIC-style two blocked kernels – can only use half the FPGA for each ⇒ ~ 50% the performance
  - Enter PR – Exploit slack by swapping kernels in the same PR region as needed

- Design: One PR region in a role and shell-esque format with blocked MMM kernel in role and SoC/DRAM logic part of the shell
Design Components

- **Vitis HLS**: MMM Kernels (Floating and Integer)
  - Implemented like Lab 3 with 3 AXI-MM interfaces to access DRAM with BRAM based scratchpad
  - Dummy kernel does nothing (same interfaces)

- **Vivado and RTL**
  - Shell created as a block design
  - RTL Top level and HLS IP wrapper as PR Partition

- **Vitis (like SDK)**: Platform exported by Vivado
  - Standalone mode (~ no OS) – embed bitstreams in program binary ⇒ bitstreams in DRAM
  - Use xilfpga library to perform PR via PCAP
MMM Kernels in HLS

- Not the best kernels – something decent and equally resource consuming
- Square block size of 64 with 16 parallel compute
  - BRAMs partitioned into 16 units

<table>
<thead>
<tr>
<th>Kernel</th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmm_float</td>
<td>28%</td>
<td>17%</td>
<td>23%</td>
<td>23%</td>
</tr>
<tr>
<td>mmm_int</td>
<td>20%</td>
<td>9%</td>
<td>1%</td>
<td>21%</td>
</tr>
</tbody>
</table>

- Integer kernel uses DSPs though in Vivado \((a*b)\)
- Dummy kernel returns size*size i.e. does nothing
Vivado Block Design

- Logic Analyzer for Debugging
- DRAM Interconnects
- Control AXI-Switch
- Decoupler
- PR Module Reset
- System LA
- AXI Interconnect
- AXI GPIO
Vitis Design

• Enable xilfpga in BSP Settings of Platform Project
  – Also change UART port for Ultra96v2 (2020 Lab 1)

• Files Documentation
  – bistream.h/s: Includes bitstream directly into the program binary transferred before every run
  – krnl_util.h/c: Functions to perform reconfiguration and running/verifying the kernel
  – main.c: Wrapper and terminal interaction
  – platform.h/c: Initializes all the hardware IPs
  – xmmm_*: HLS generated driver files to communicate with the HLS IPs with nice functions
What you’ll need to do...

- Local Vitis Installation to connect to Ultra96v2
- Ultra96v2 connected to your PC using the JTAG-to-UART extension – refer 2020 Lab 1 for more
  - Set Ultra96 to JTAG mode (pair of tiny switches)
  - Linux requires granting permissions for ports
- Start Vitis with workspace as pr_example.vitis
  - Build and “Run As” with the first configuration
  - Open Vitis Serial Terminal (any serial monitor will do) and connect to the board as you wait for the program binary to be transferred (~ 2 min)
  - Go nuts!

Let’s see it in action...
Example Archive Documentation

- `mmm_hls`: Contains 3 HLS projects for the three types of kernels (Note: The top-level files from the float and int kernels have been modified to standardize interfaces)
- `pr_example.srcs`: The import folder within contains sources from HLS, the top-level wrapper, and the HLS wrapper snapshotted during export
- `pr_example.vitis`: Vitis workspace containing the top platform project and `mmm_system` application project and relevant source files
- `pr_example.runs`: Contains outputs of all Vivado runs including the bitstreams referenced in the Vitis project
- `pr_example.xpr`: Vivado Project file
Known Shortcomings

• Bugs:
  – GOPS calculation uses integers to get operation count and can overflow for large matrices
  – Error printing for integer kernel uses "%f" in printf causing erroneous casting from int to float

• Enhancements:
  – Fabric is clocked at 100 MHz and kernels designed with a target of 200 MHz – can increase clock frequency to double performance
  – If compiled bitstreams are not changing, iterating through software takes long – store bitstreams in SD Card instead and use xilffs to read
  – Check for size not being a multiple for block size