Demand-Driven FPGA Configurations Using PR, Linux, and PYNQ

Jeff Goeders, Tanner Gaskin, and Brad Hutchings
Dept. of Elec. and Comp. Eng., Brigham Young University, {jgoeders, brad_hutchings}@byu.edu

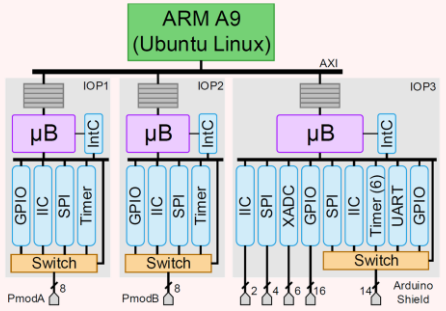
Overview

- Leverage partial reconfiguration to build PYNQ overlays on demand.
- "Software Library" approach to hardware features:
  - Collection of hardware modules loaded on demand.
  - Support graceful Linux driver load/unload.
- Allow average PYNQ users to leverage FPGA reconfigurability.
- ***Create custom hardware configurations without hardware expertise***.

Proof of Concept – PYNQ I/O System

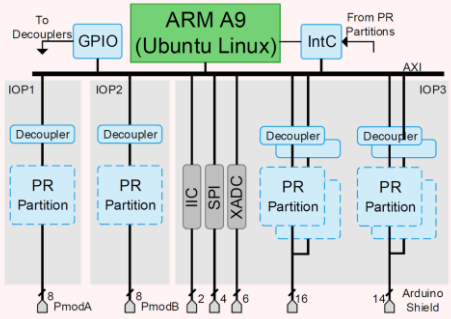
The Original PYNQ I/O System

- "Kitchen sink" approach with many HW modules.
- Fixed configuration; any changes require a full recompile.



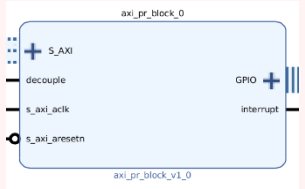
Our PR "On-Demand" Design

- 6 PR regions.
- Any region can do any protocol.
- New modules added as needed.
- 20% reduction in area.



Design Flow

- Custom IP to add PR functionality to your PYNQ overlay
  - AXI memory-mapped connection
- Integrated with the PYNQ framework
  - Python drivers loaded automatically
  - Kernel drivers also supported
- Publicly available at https://github.com/byuccl/pynq_prio_pip.git
  *current work not fully released.



PR & Linux Drivers

Device Tree Overlays

- Device tree fragments can be added to the tree at run-time; essential for supporting a PR framework.

User Experience

- We provide a library of classes that abstract away the PR and device tree modifications from the user.
- User can make a single call to reconfigure a region.
- The driver for the old module is gracefully removed and the new driver is automatically loaded.

Device Tree Fragment

```
...
pr0: pr0@41A10000 {
  compatible = "generic-uio";
  reg = <0x41A10000 0x10000>;
  interrupt-parent = <&xil_intc>;
  interrupts = <0x0 0x00>;
};
pr1: pr1@41A20000 {
  compatible = "generic-uio";
  reg = <0x41A20000 0x10000>;
  interrupt-parent = <&xil_intc>;
  interrupts = <0x1 0x00>;
};
pr2: pr2@41A30000 {
...
```

Sample User Application Code

```
from pynq.overlays.io_pr import IoPr
from pynq.overlays.io_pr.drivers.uart import Uart
from pynq.overlays.io_pr.drivers.gpio import GPIO

# Instantiate IoPr class to interact with overlay
#   - Use user space drivers
overlay = io_pr.IoPr(driverModeKernel = False)

# Program RP0 with a UART
overlay.rp0.configure("uart")
uart = Uart(overlay.rp0)

# Sent data out over the UART
uart0.sendData([0xDE,0xAD,0xBE,0xEF])

# Program RP2 with GPIO
overlay.rp2.configure("gpio")
gpio = GPIO(overlay.rp2)

# Set GPIO as outputs and turn on LED0
gpio.setDirection(GPIO.ALL_OUTPUTS)
gpio.setValue(0x01)
```

Benefits

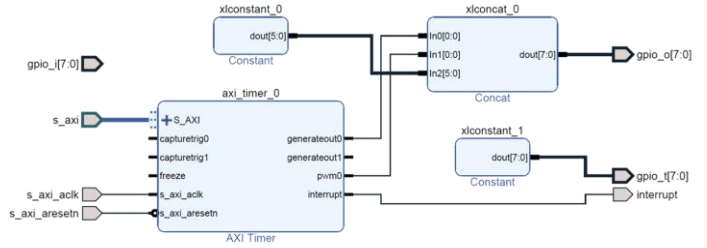
On-Demand Scalability

- Library of modules are given to the user
  - 100s of modules is feasible
- Library can include different configurations of an IP
- User chooses at run-time which hardware controllers will be present in the system.

Simplified Hardware Design

- Each module is a stand-alone Vivado project
- Faster compile (5-10 min vs 1 hour)
- Static design remains locked; no redoing timing closure
- Gradual introduction to HW design for educational use

Vivado Project for Timer PR Module



Conclusions

- Proposed framework of PR-based bitstream assembly provides several advantages while maintaining a simple HW/SW design flow.
- With transitions to larger devices (PYNQ on UltraScale+) the need for a scalable, modular approach will only increase.