18-643 Lecture 14: FPGA Programmability

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today: appreciate FPGAs as truly programmable and dynamic devices

• Notices
  – 10/15: Midterm (cover through L13)
  – 10/17: Eriko Nurvitadhi (Intel) guest lecture
  – 10/21 (due noon): Lab 3 AND proposal slides
  – 10/{24, 25, 26}: proposal presentations

• Readings
  – Ch 4, Reconfigurable Computing
  – Section 5.6, The Zynq Book
18-643 Review Template: Summary

• What to look for in a paper/presentation
  – what is the question/problem?
  – why is this question/problem important?
  – why is this question/problem hard?
  – what is the answer/solution offered by the paper?
  – what is new, novel about the answer/solution?
  – how does the paper argue/support the answer/solution is correct/good?

• If you don’t know the answers, you didn’t “read” the paper, OR, the paper/presentation is bad
18-643 Review Template: Critic

• What to think about while reading
  – soundness: is the paper's answer/solution correct/good?
  – impact: is the paper's answer/solution important?
  – novelty: does the paper teaches something new and not obvious?
  – strengths: what makes the paper standout?
  – weaknesses: what could be improved?

• If you don’t know the answers, you didn’t “read” the paper

Don’t accept what a paper says unless you agree
1980’s Xilinx LUT-based Configurable Logic Block (in a sketch)

- 2 functions \((f \text{ & } g)\) of 3 inputs OR 1 function \((h)\) of 4 inputs
- hardwired FFs (too expensive/slow to fake)
- Just 10s of these in the earliest FPGAs
Configurable Routing
(1980s Xilinx simplified)
How about no mask, no fab? i.e., “field programmable”

• Again, mass produce identical devices but this time fully-finalized

• Then what can be changed?
  – SRAM           EPROM        (anti)fuse
    \begin{align*}
    \text{bits} & \{1,0\} & \{1,0\} & \{1,0\} \\
    \text{connections} & \{1,0\} & \{1,0\} & \{1,0\}
    \end{align*}

  – pass gate       mux         diode
    \begin{align*}
    A & \quad \quad \quad \quad \quad \quad B \\
    A & \quad \quad \quad \quad \quad \quad B \\
    A & \quad \quad \quad \quad \quad \quad B
    \end{align*}
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
Setting Configuration Bits

- Behind-the-scene infrastructure
  - doesn’t need to be fast (usually offline)
  - simpler/cheaper the better (at least used to be)

- Could organize bits into addressable SRAM or EPROM array
  - very basic technology
  - serial external interface to save on I/O pins
Serial Scan

• SRAM-based config. bits can be setup as one or many scan-chains on very slow config. clock
  – no addressing overhead
  – all minimum sized devices

• At power-up config manager handshake externally (various options, serial, parallel ROM, PCI-E, . . . )

Full-fledged config. “architecture” in modern devices to support scale and features
Modern Configuration Architecture
e.g., Stratix® 10 Secure Device Manager

- triple-redundant secure processor
- each sector managed by its own processor

[Figure 2: “Intel® Stratix® 10 Secure Device Manager Provides Best-in-Class FPGA and SoC Security”]
“Field (Re)programmable”

• Programmability is a very costly feature
• When we wanted FPGA to be ASIC
  – programmability avoided manufacturing NRE
  – programmability reduces design time/cost
    (incremental development; at speed testing; field updates, etc.)
  – BUT once programmed at power-on, FPGA is fixed
• Let’s use programmability to be more than ASIC
  – repurpose fabric over time, at large and small time scales
  – share fabric by multiple applications concurrently
Turn Programmability into Performance

• Amdahl’s Law: \( S_{\text{overall}} = \frac{1}{(1-f) + \frac{f}{S_f}} \)

• \( S_{f-\text{ASIC}} > S_{f-\text{FPGA}} \) but \( f_{\text{ASIC}} \neq f_{\text{FPGA}} \)

• \( f_{\text{FPGA}} > f_{\text{ASIC}} \) (when not perfectly app-specific)
  – more flexible design to cover a greater fraction
  – reprogram FPGA to cover different applications

[based on Joel Emer’s original comment about programmable accelerators in general]
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
- Basic technology mature but usage not prevalent under the ASIC model
- Essential to FPGA as a flexible, sharable computing device
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

Static partition: top

reconfig. partition:
instance_foo

reconfig. partition:
instance_bar

m1

m2

m3

m4

m5
Concrete Syntax (Xilinx’s approach)

module top();

....

foo instance_foo (a1, a2, ...);
bar instance_bar (b1, b2, ...);

....
endmodule

module foo(
  input a1, a2, ... 
  output ax, ... )

// nothing here
endmodule
Implementation Flow

1. Designate `instance_foo` and `instance_bar` as Reconfigurable Partitions (RPs)
2. Bind `instance_foo` and `instance_bar` to their most resource-demanding variants (e.g., `m2` and `m4`)
3. Floorplanning
   - draw bounding boxes for RPs
   - reserve enough resource for largest variant
   - place partition interface pins
4. Place-and-route full design with `m2` and `m4` in RPs
   - extract partial designs of just `m2` and `m4` RPs
   - extract static partition (full design with blank RPs)
What this looks like . . .

[Vivado Implementation Screenshot]
Implementation Flow (continued)

(Re-start from static partition and locked floorplan)

5. Bind `instance_foo` and `instance_bar` to remaining variants (e.g., `m1` and `m5`)

6. Place and route new full design

7. Extract partial designs of just `m1` and `m5` RPs
   
   . . . repeat for `m3` . . .

End Result

- 1 full-design bitstream (including `m2` and `m4`)
- partial design bitstreams individually for `m1`~`m5`

Yes, this flow is a little clunky (see UG947 to try)
At Run Time

• Power up with full-design bitstream
• Partial bitstreams in DRAM or flash memory
• Configuration API driven by ARM or fabric
  – reconfig. time depend on size, as low as msec
  – handshake signals to pause/start partition interface
Today’s Practical Constraints

• Number and size of PR partitions fixed apriori
  – too few/too large: internal fragmentation
  – too many/too small: external fragmentation

• Not all PR partitions are equal—even if same interface and shape
  – a module needs a different bitstream for each partition it goes into
  – build and store upto MxN bitstreams for N partitions and M modules

• PR is not all that fast . . .
Today’s PR Overhead

• Reconfigurations take on the order of msec
• Time to reconfig grows with PR partition size (~128MB/s with Xilinx PCAP)
• Only one PR with PCAP at a time
Use Case: Vision Processing Pipeline

vision stage IP library + pipeline specifications

runtime

embedded ARM core
  
scheduler
  
mapper (module to RP)
  
Interconnect and DMA configurer

FPGA
  
DPR
  
DPR
  
... DPR
  
Programmable Crossbar
  
DMA
  
DMA
  
DMA
  
IO

plug-and-play architecture

Use Case: Vision Processing Pipeline

vision stage IP library + pipeline specifications

runtime

embedded ARM core
  
scheduler
  
mapper (module to RP)
  
Interconnect and DMA configurer

FPGA
  
DPR
  
DPR
  
... DPR
  
Programmable Crossbar
  
DMA
  
DMA
  
DMA
  
IO

plug-and-play architecture
Spatial and Temporal Multitenancy

Camera → M_A → M_B → M_C → Display
Camera → M_D → M_E → M_F → Display
Camera → M_G → M_H → M_I → Display

ARM core
(user code + SW management)

Flash
DRAM
AXI-PCAP Bridge
AXI Master Interface
FIFO
PCAP Interface

Overlay Crossbar
DMA
Camera
Display
Time-Multiplexing Feasibility [FPL2018]

- Interleaving (a) 2 pipelines and (b) 3 pipelines
- Pipelines differ between 1 and 6 PR partitions

It takes more than doing PR in quick successions!!!
Cost and Energy/Power Benefits [FPL2019]

- Application case study with 6 modules
  - color-based detector triggers follow-up processing
  - meets throughput and latency requirements
  - ~3x logic saving (7x $ saving in parts cost)
  - ~30% power/energy saving in worst case

"large" FPGA
- detect
- stereo
- SIFT
- etc.

"small" FPGA
- DPR

Static Mapping

Timeshare
Greater break from ASIC mentality

- **Dynamism** — actually use the programmability
  - support more functionality on same parts cost
  - achieve better performance by specializing
- **Shareability** — multitenancy to consume “slack”
  - too much logic: partition fabric spatially
  - too much throughput: repurpose fabric temporally
- **Manageability** — bring FPGA under OS purview
  - part of compute resource pool (CPU cycles, DRAM)
  - seamless interface, virtualization and isolation (security and QoS)

**Dynamic Partial Reconfiguration is a key capability**
Parting Thoughts

• FPGA’s win over processor is speed and efficiency; FPGA’s win over ASIC is flexibility
• For computing, don’t use FPGA like an ASIC; but don’t think about it like a processor either
• Partial reconfiguration really does work!!
  – could still be much better . . . .
  – have to find strong new uses and use modality
  – have to properly integrate and support it (design tools, scheduler, virtualization/protection, QoS ...)

Go from applying FPGAs to computing
⇒ making better FPGAs for computing