



*Reconfigurable Computing:
This is Not Your Father's x86*

Herman Schmit
Dept. of ECE
Carnegie Mellon University



Interesting Time

- ◆ Huge numbers of transistors
- ◆ Changing nature of applications
 - Media-centric
 - Real-time
 - Continuous data streams
 - ◆ Embarrassing amounts of parallelism
 - ◆ Small data elements (1 - 16 bit)
- ◆ Time to reconsider everything

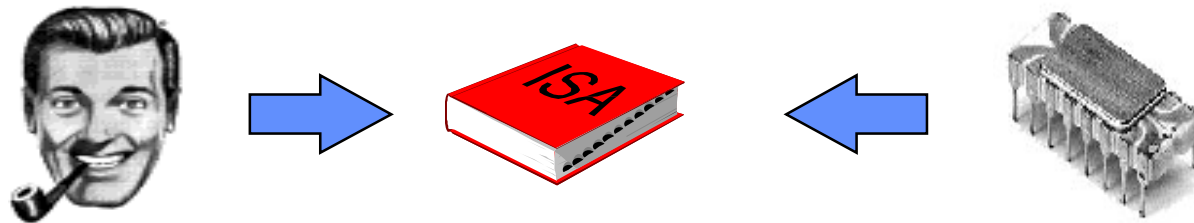
Let's reconsider the computer

◆ Instructions

- Enabled re-use of fixed hardware resources
- Sequence hardware through data (instructions)
- Enabled compilers, languages

◆ What is an ISA?

- Interface (contract) between user and hardware

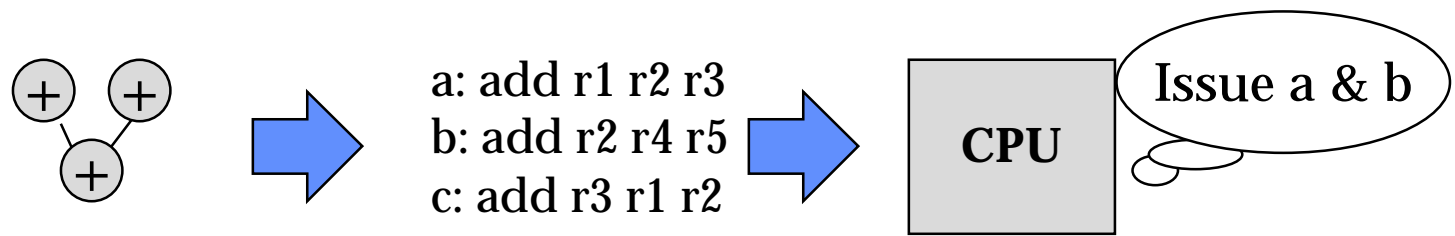


Whither the ISA?

- ◆ Original Purpose Irrelevant
 - Most developers don't see ISA anymore
 - ISP don't "look" like their ISA anymore
- ◆ What are they still good for?
 - Contract between compiler and hardware
 - Contract from one CPU generation to another
 - ◆ Forward compatibility
- ◆ What's wrong with ISAs?

Why get rid of instructions?

- ◆ Forced sequentialization of algorithms



- ◆ Operands routed through “registers”
 - Name dependencies: register renaming
 - Pipeline stalls: Forwarding paths
 - Registers are global entities
 - ◆ No way to keep operations physically close

Development Cost of ISPs

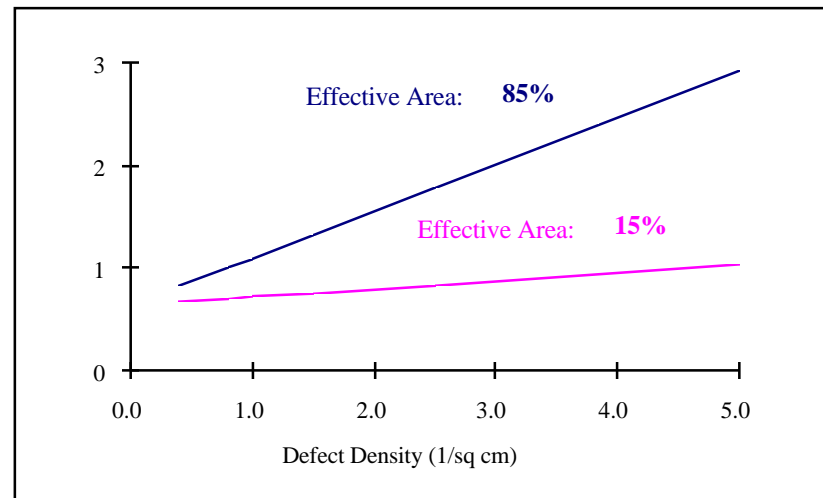
- ◆ Motorola 68K: \$5M (1979)
- ◆ Intel Pentium: \$250M (1993)
 - Not including FDIV bug!
- ◆ Growing at 25% per year:

\$6B in 2007!

- ◆ Sun heuristic:
n% performance feature must be verified in *n* weeks

Manufacturing Cost of ISPs

- ◆ CPUs have high “effective area”
 - 85% (Pentium) - 97% (PowerPC 601)
- ◆ Not a problem - if defect densities stay low
- ◆ What if they don't?



Die area: 2.56 sq. cm
Dingwall's equation

Software without Instructions

- ◆ Design “hardware” for each application
- ◆ Map it onto programmable hardware:
 - Erase it when your done: **reconfigurable**
 - Programming by **interconnect** not **instructions**
- ◆ Pros:
 - No “instruction” bandwidth
 - No overhead for determining parallelism
 - Increase freedom and flexibility

Performance!

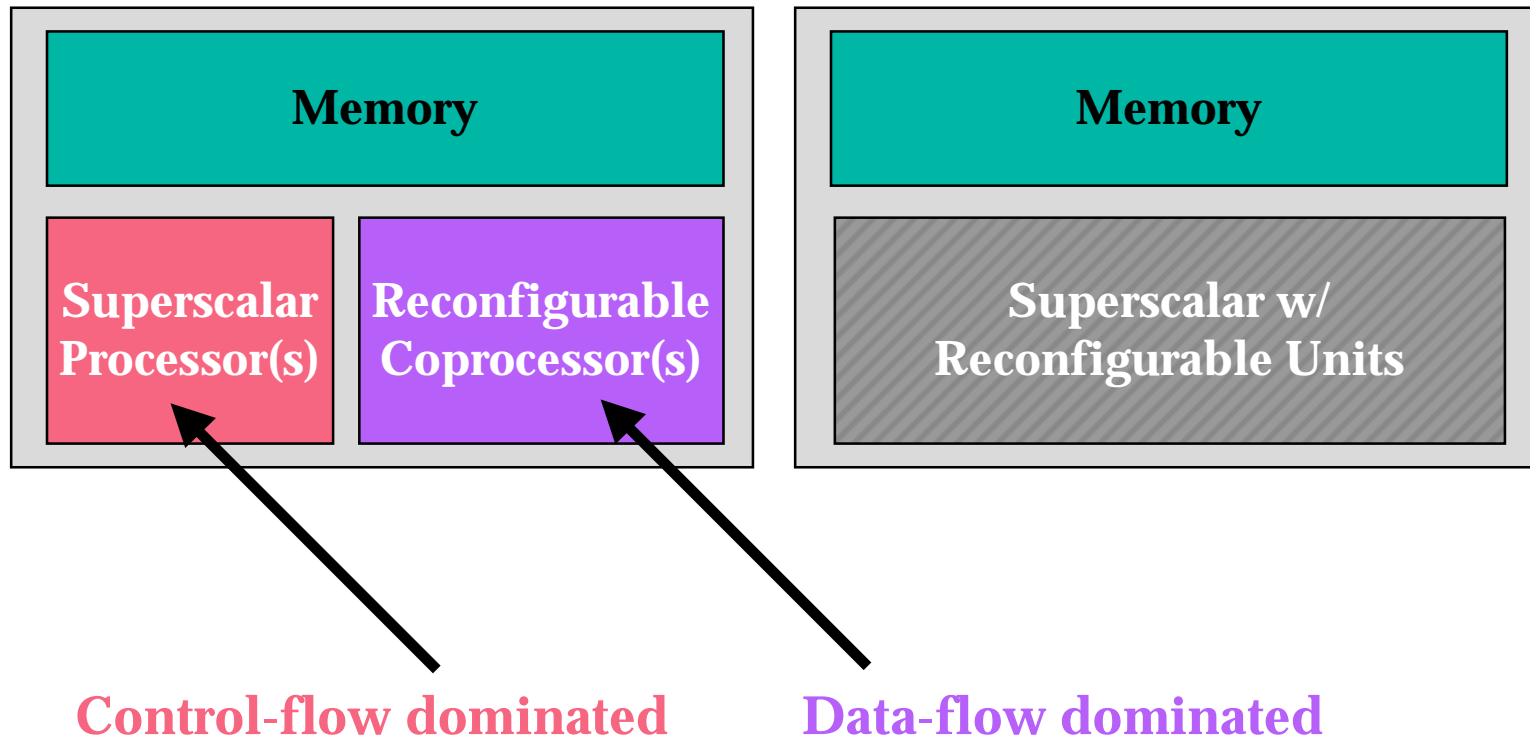
Performance

| Algorithm | FPGA system | Comparison CPU | Speedup |
|------------------------|------------------|------------------------|-------------|
| DNA Matching | SPLASH 2 | SPARC 10 | 4300 |
| Stereo Vision | PARTS | Generic RISC 400MHz | ~30 |
| RSA Crypto | PAM | Alpha 150MHz | 17.8 |
| Ray Casting | RIPP-10 | Pentium 75MHz | 33.8 |
| FIR filter | 1 Xilinx FPGA | DSP 50MHz | 17.9 |
| Hidden Markov Model | 1 Xilinx FPGA | SPARC 10 | 24.4 |

More Pros

- ◆ Local connections:
 - No central bottleneck (register file or memory)
 - Dependent operators can be close
 - Local clocks possible
- ◆ Regular structure:
 - Testability
 - Fault tolerance through redundancy
 - ◆ Increased yield, decreased cost
 - ◆ In-the-field reconfigure around hard faults
 - Low development cost

Microprocessor 2007



Potential Roadblocks

- ◆ Cost - Performance
- ◆ Forward-compatibility
- ◆ Compilers
- ◆ Interfaces with CPUs
- ◆ Mind-set

Cost and Performance

Reconfigurable vs. ISP

(Baseline: Application in full-custom silicon)

$$\frac{\text{FPGA Cost}}{\text{FPGA Performance}} \leq \frac{\text{ISP Cost}}{\text{ISP Performance}}$$

$$\frac{\text{Yield Benefit} \cdot \text{Area Increase}}{\text{Performance Reduction}} \leq \frac{\text{ISP Area Increase}}{\text{ISA Performance Penalty}}$$

$$\frac{\frac{1}{2} x \cdot 20 x}{\frac{1}{4} x} \leq \frac{\text{ISP Area Increase}}{\text{ISA Performance Reduction}}$$

$$\text{Customization} \geq \frac{40 x}{\text{ISP Area Increase}}$$

Customization Performance

- ◆ Right Granularity
- ◆ Right Parallelism
- ◆ Right Pipelining
- ◆ Right Operations
- ◆ Constant Propagation

Granularity and Parallelism

- ◆ CPUs (pre-multimedia)
 - Wasted functional units: 32 bit add for 8 bits
- ◆ Post-multimedia
 - Data alignment and utilization
 - Clock cycle based on long word
 - ◆ time (64 bit op) >> time (8 bit op)
 - ◆ one half to one eighth the throughput

Pipelining

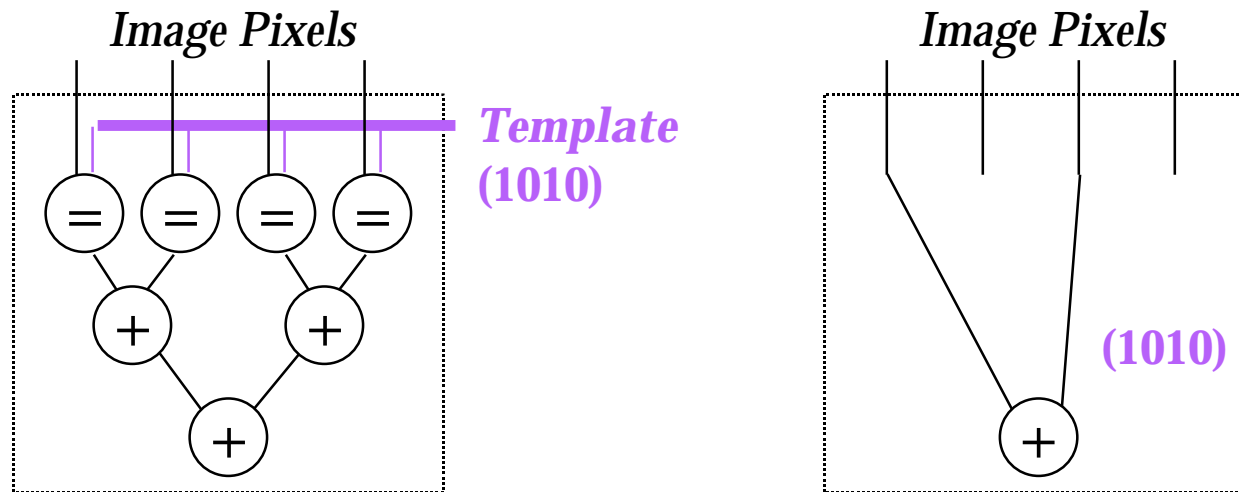
- ◆ Non-vector CPUs:
 - Operation latency is bad
 - Low degrees of operation pipelining
 - ◆ e.g. Multiply = 2 cycles
- ◆ FPGAs:
 - Pipeline at every gate: up to 250 MHz
 - When operation latency is bad, don't pipeline

Right Operations

- ◆ Bit reversal:
 - Cryptography and FFT addressing
- ◆ Complex functions:
 - $(a \& b) | (a \wedge c)$
- ◆ Small parallel look-up tables
 - Cryptography
 - Vector quantization
 - Image compression

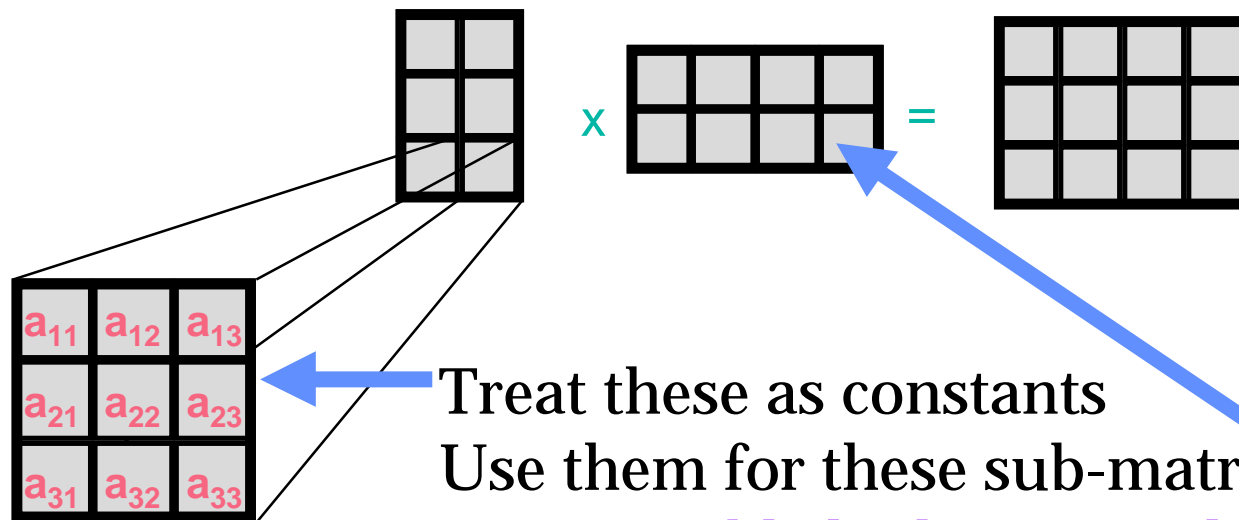
Constant Propagation

- ◆ Filter coefficients:
 - multiplication -> shift and add
- ◆ Encrypt/decrypt for **one** key
- ◆ Pattern match for **one** template



Not-so-constant Propagation

- ◆ Adaptive filters
 - Adaptive layer reconfigures hardware
- ◆ Matrix Multiplication:



Treat these as constants
Use them for these sub-matrices

Fastest published RSA implementation

Customization Performance

- ◆ Right Granularity: 1.5x
- ◆ Right Parallelism: 1.5x
- ◆ Right Pipelining: 10x
- ◆ Right Operations: 2x
- ◆ Constant Propagation: 5x

Customization up to 200x benefit!

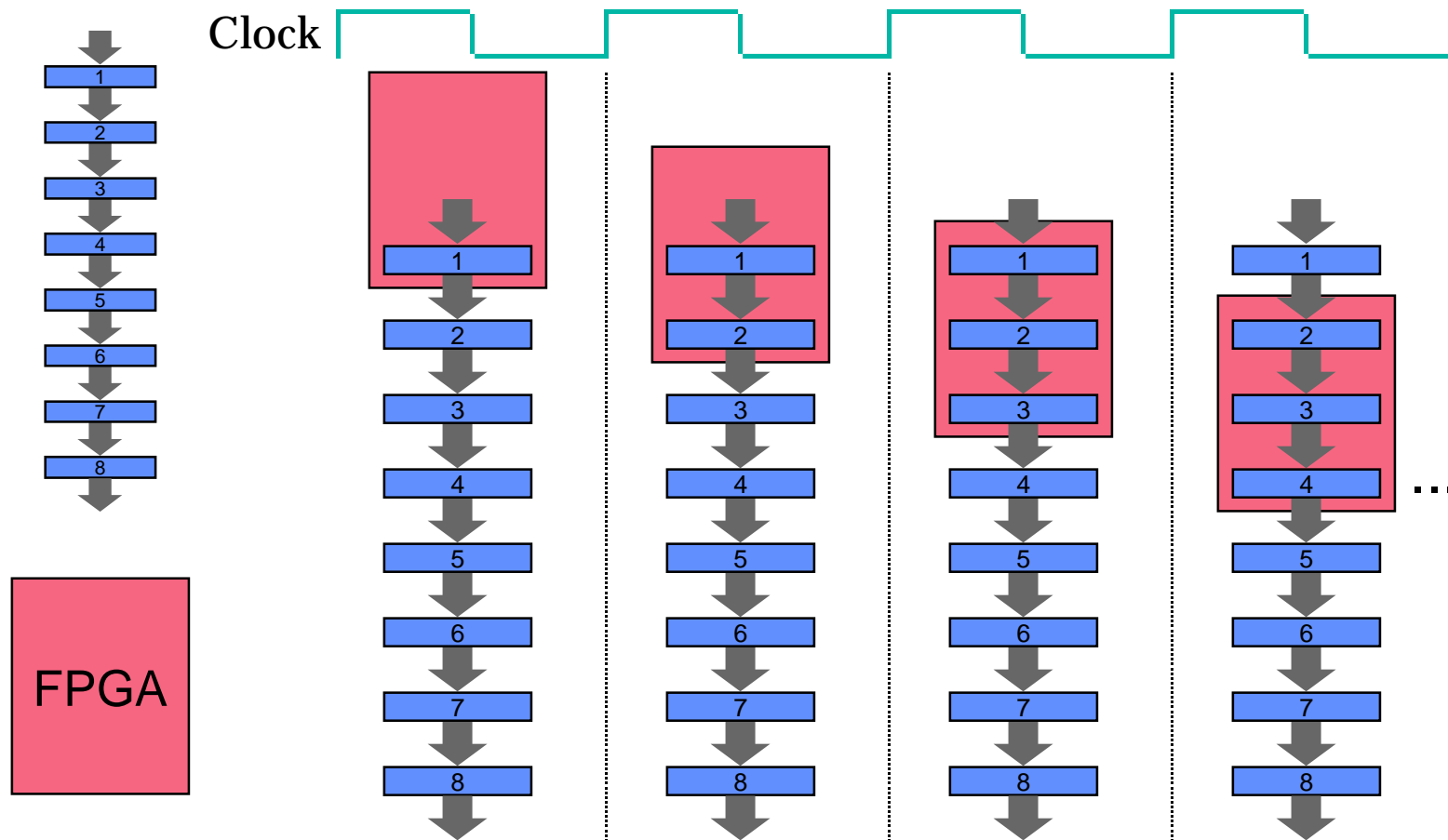
Forward Compatibility

- ◆ ISA: contract between silicon generations
- ◆ Reconfigurable: parallelism is **fixed**
 - Faster silicon -> more performance
 - More silicon -> nothing
- ◆ Solution:
 - Design for future (**huge!**) silicon
 - Time-multiplex existing hardware
 - Heavily pipelined applications

Time-Multiplexing

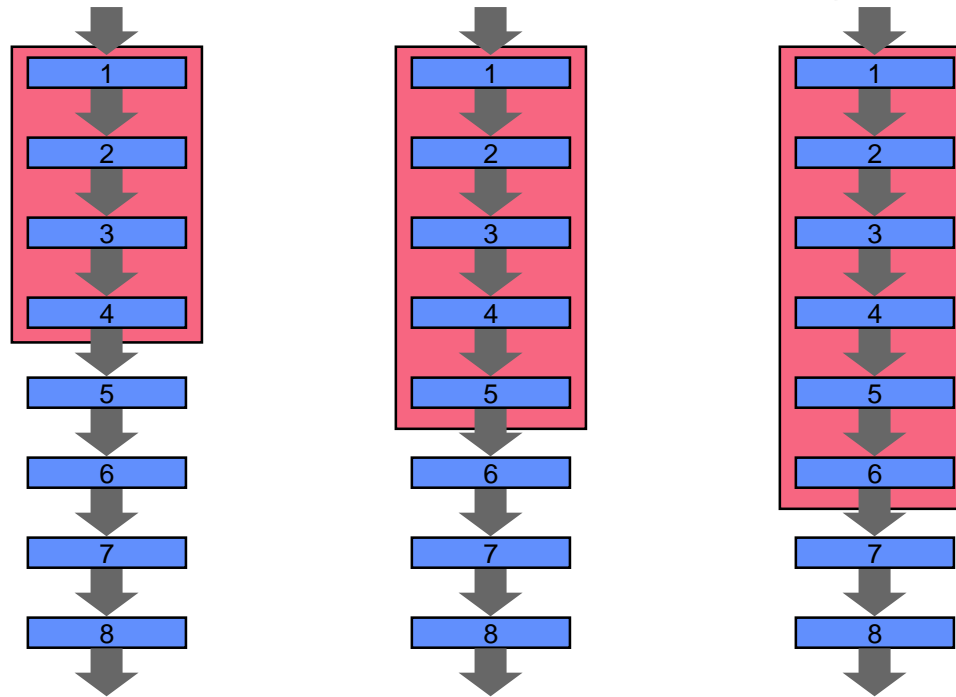
- ◆ Break app up into pieces:
 - Pipeline pieces = stages
- ◆ Create configuration info for each stage
- ◆ Load a stage in every cycle
 - Pipeline the configuration process
 - Pipelined Reconfiguration

Pipelined Reconfiguration



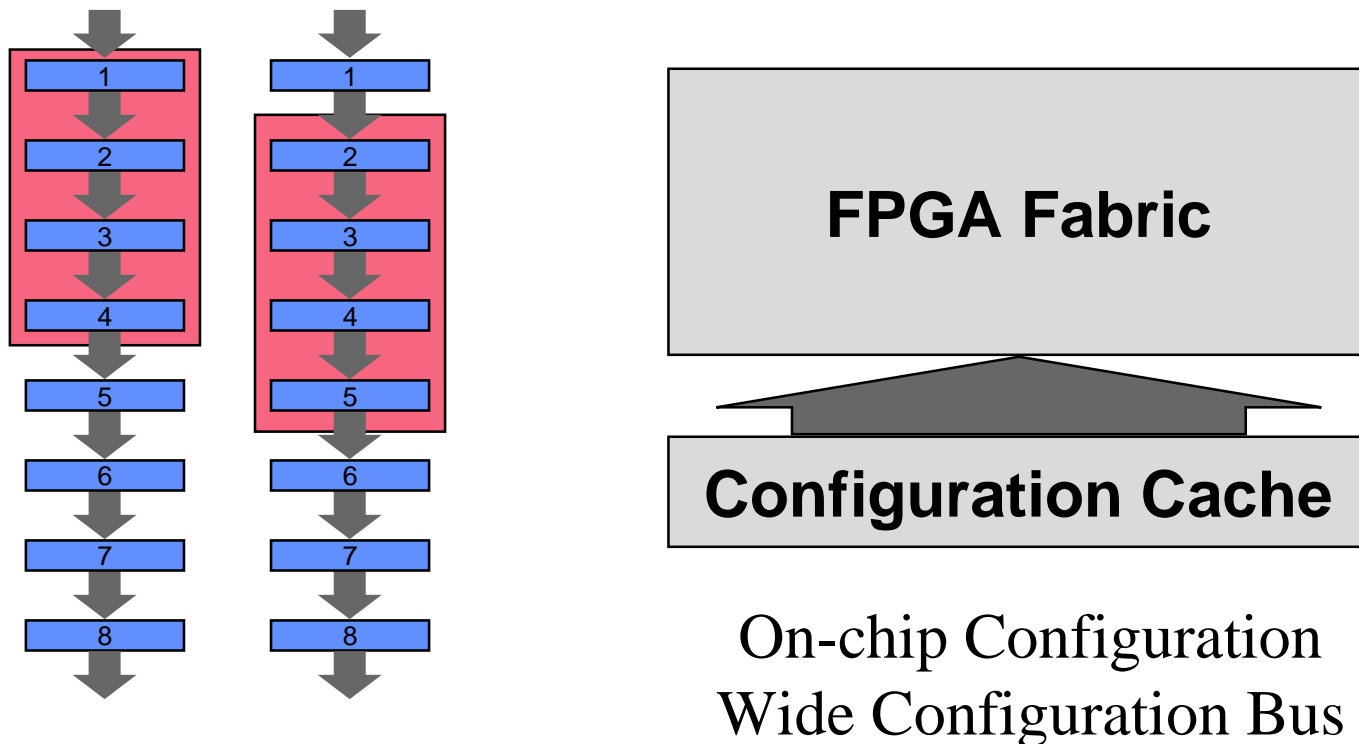
Forward Compatibility

- ◆ More hardware = more throughput

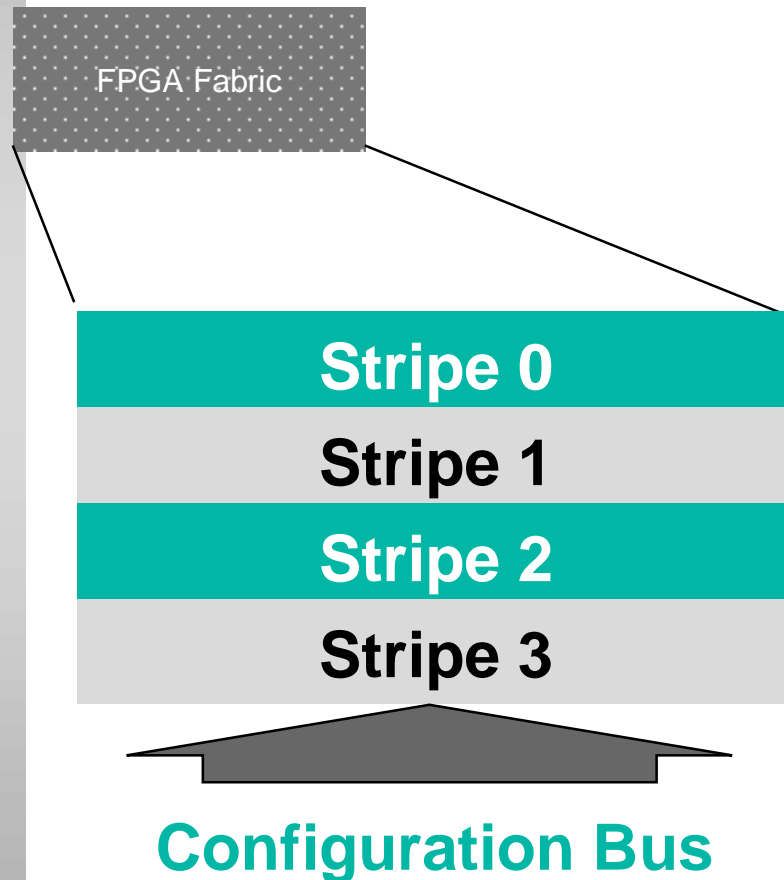


PipeRench Architecture

- Objective: Configure 1 stage per cycle



PipeRench Architecture



The Stripe:

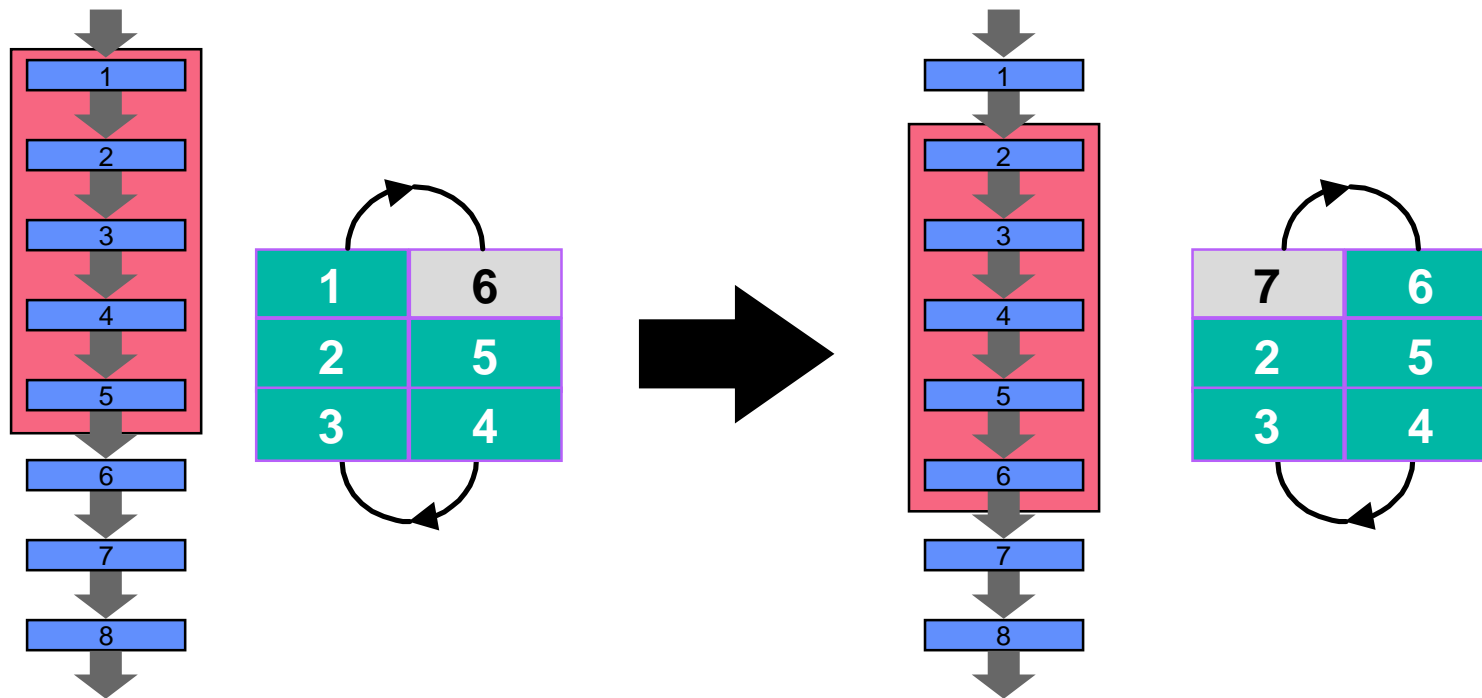
Configured in one cycle

Local, registered interconnect
between adjacent stripes

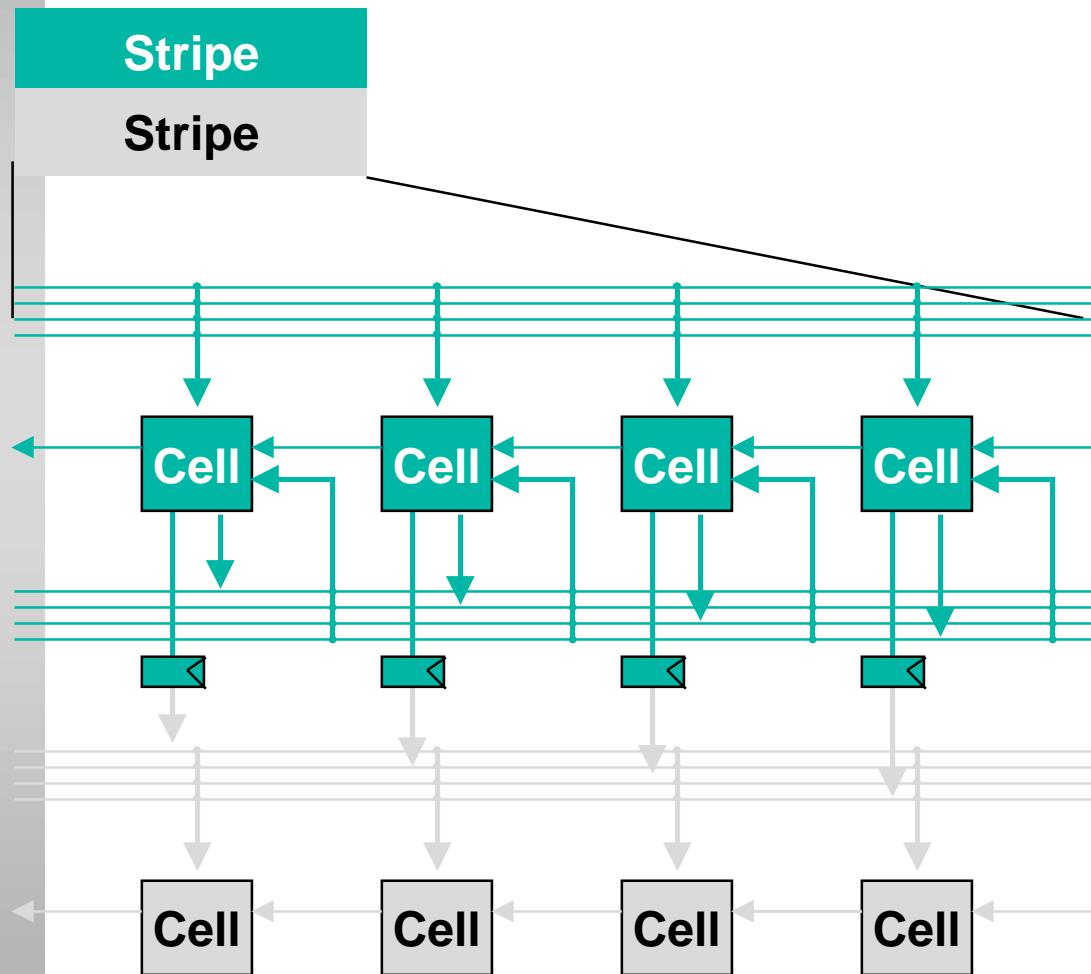
Designed to implement one
pipeline stage

Ring Interconnect

- Configuration doesn't really move!
 - Ring: local interconnect, stationary configuration



Stripe Architecture



All Cells/Busses: 4-bits

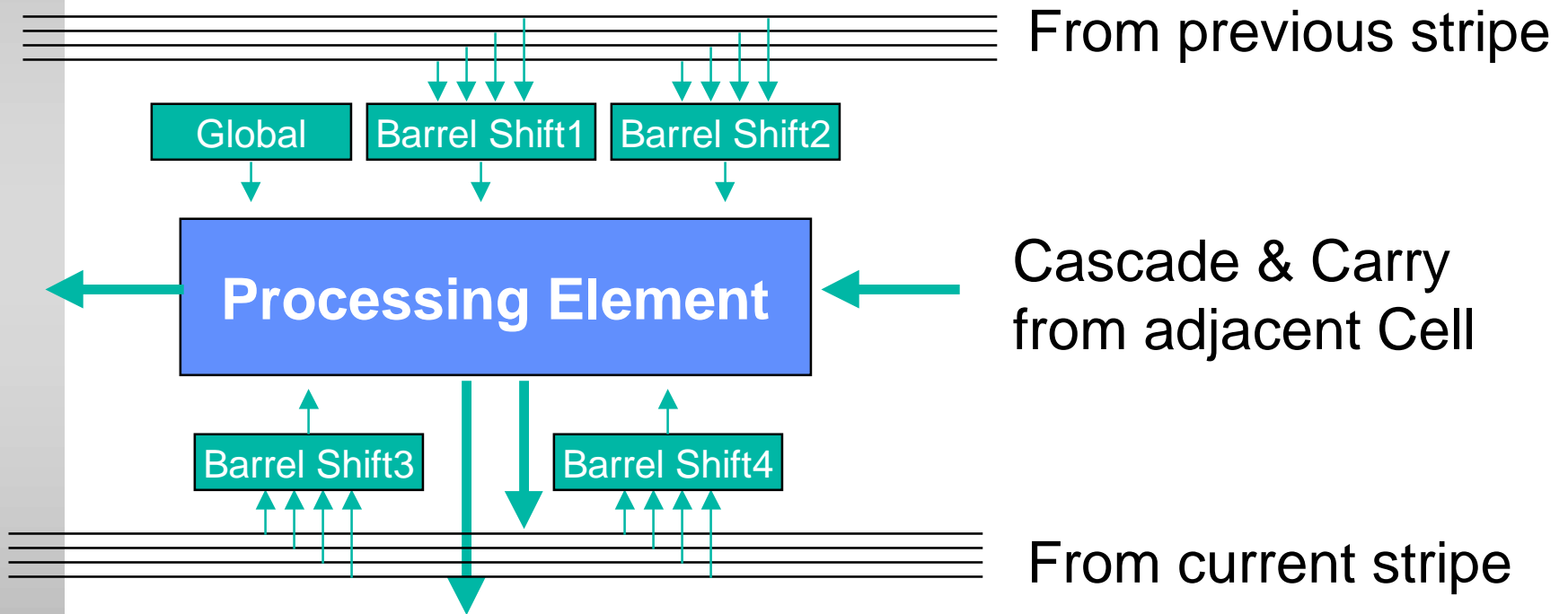
Reduced config. info

Carry and cascade
paths

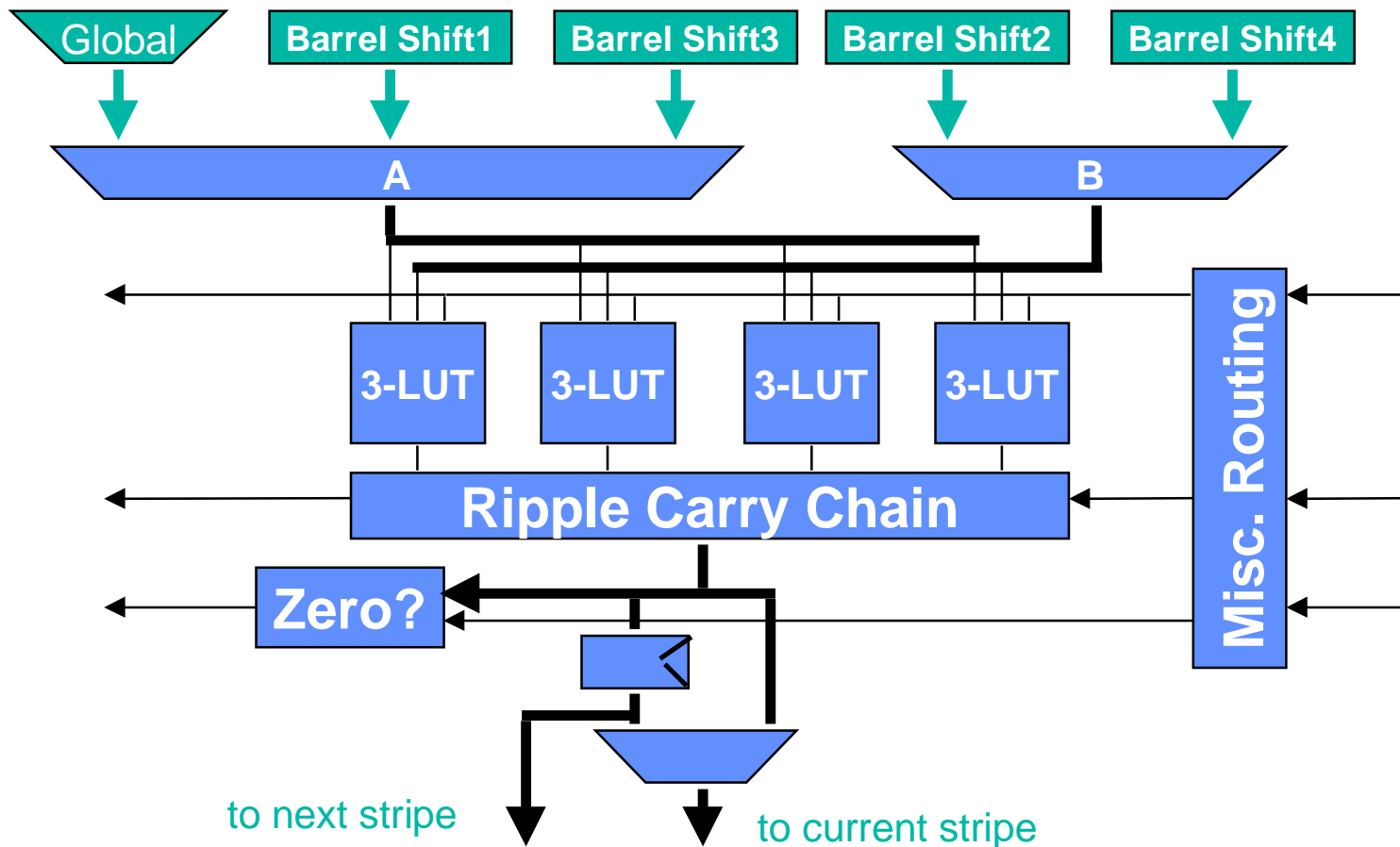
Intra-stripe interconnect:

MUX-based

Cell Architecture



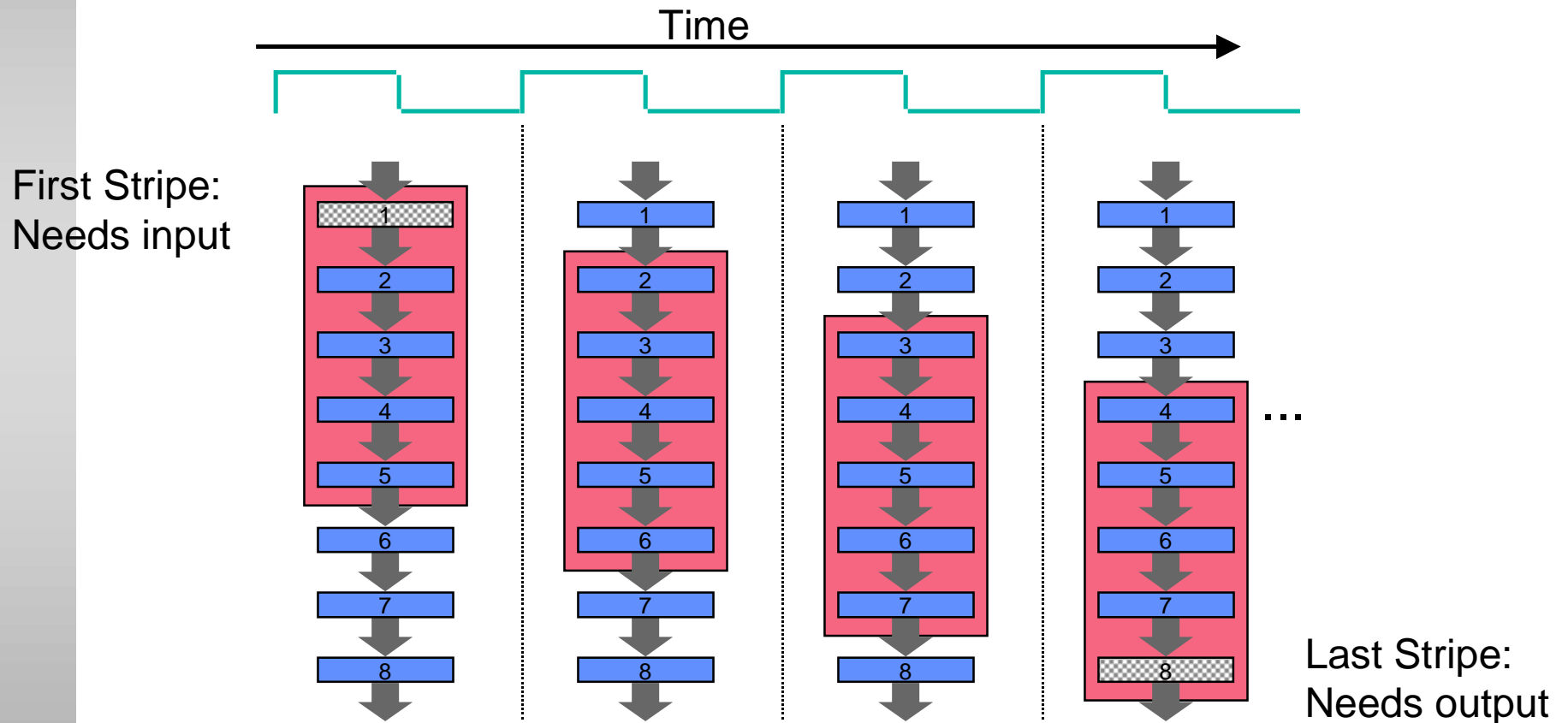
Processing Element



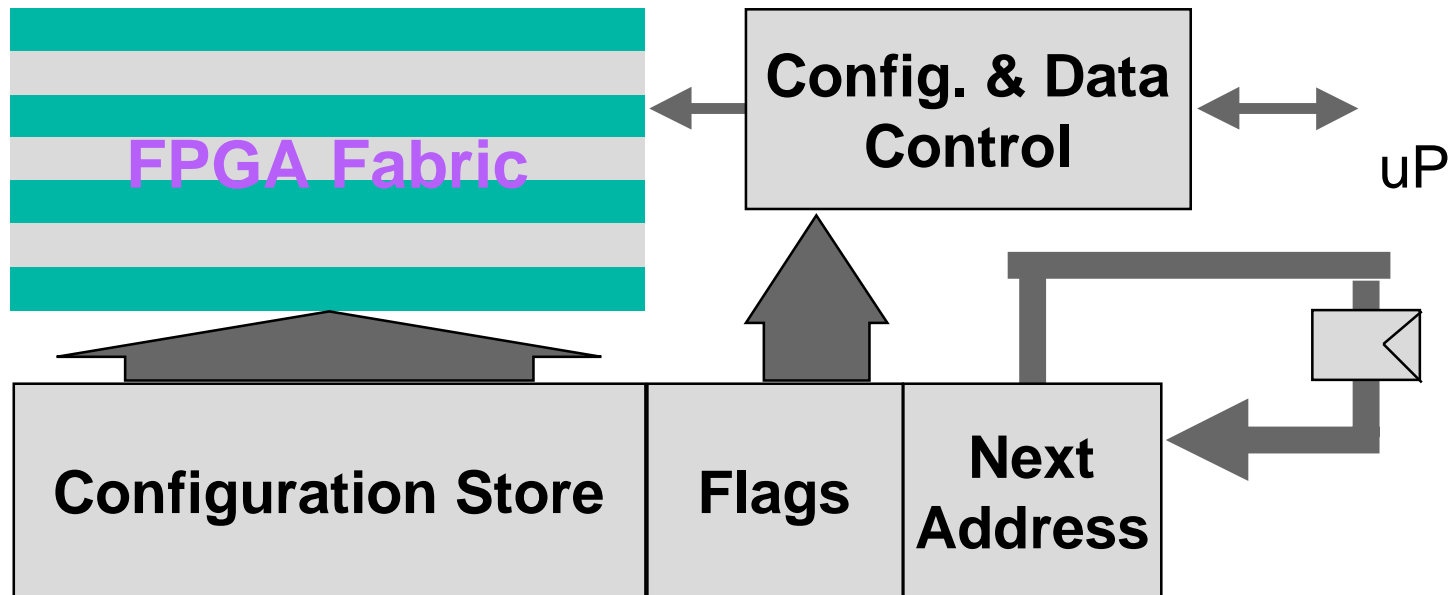
Configuration Control

- ◆ Determine whether virtualization needed
- ◆ Manage configuration memory
 - Load applications from outside
- ◆ Store and restore state information in stripe
- ◆ Control global resources
- ◆ Provide control interface

Memory Access Control

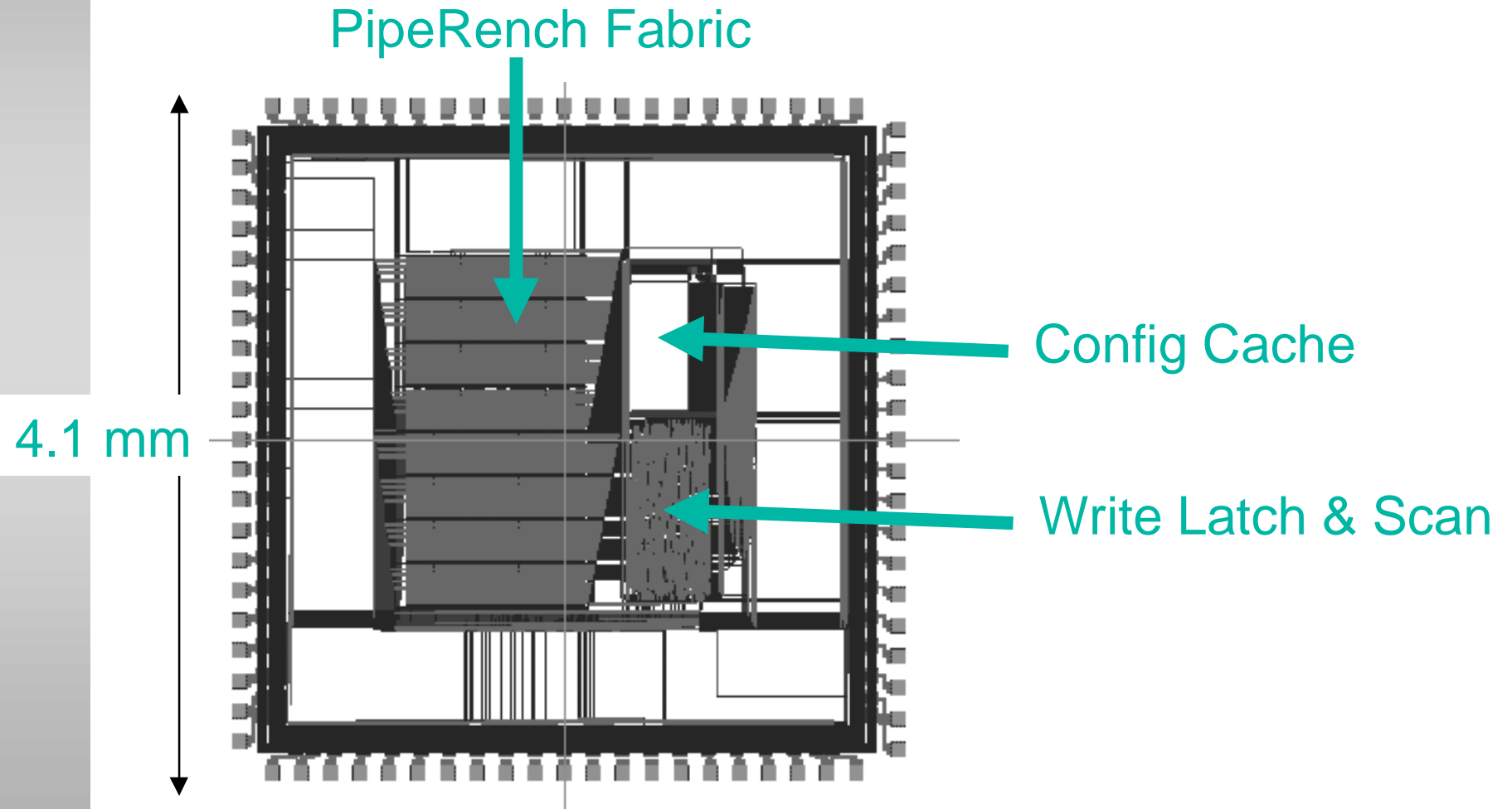


Configuration Control



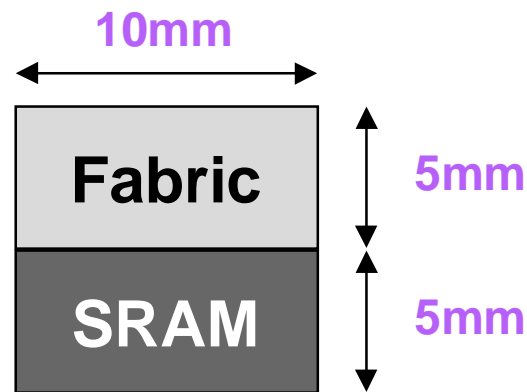
Flags:
Write bus (4)
Read bus (4)
Store/Restore
End

PipeRench Prototype



PipeRench Roadmap

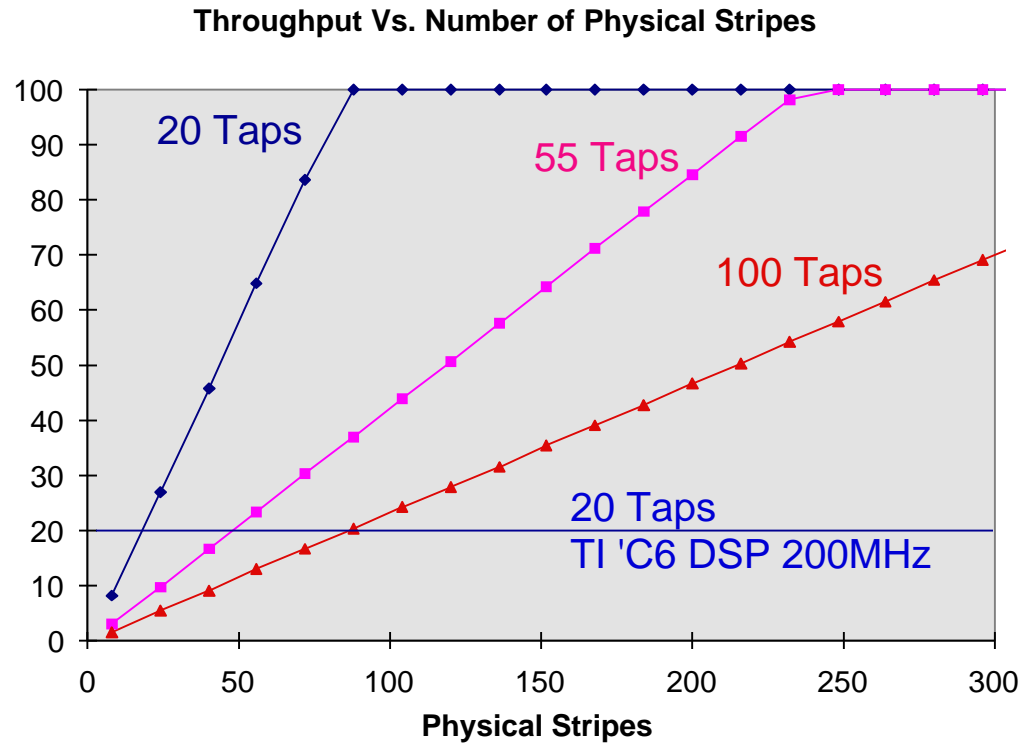
■ Floorplan:



- 28 Stripes (in 0.35 μ m 4 metal technology)
- 32 Cells in each stripe (128 bit data path)
- 512 Stripes of configuration cache
 - ✦ Virtual Memory stores 18 times the real fabric

■ Target speed: 100MHz

FIR Filter Performance



Older technology, 2x performance, and scaleable.

IDEA Encryption Performance

- ◆ Symmetric encryption for PGP
- ◆ Key-specific hardware
 - Generate configuration on-the-fly
- ◆ 232 Virtual Stripes, ~ 64 bits width
 - 940 Mb / sec with 28 rows @ 100MHz
 - ASIC: 177 Mb / sec (1 cm², 1.2 mm, 25 MHz, 1993)
 - ASIC: 2 Gb / sec (today)
 - Pentium: ~1Mb / sec
 - Scales to 7.8 Gb / sec with 232 rows

Benefits for Compilation

- ◆ **Problem: Hard constraints**
 - Need one more wire**
 - Need one more cell**
- ◆ **Expectation: Robustness**
 - CC never says “This legal C won’t compile.”**
- ◆ **Solution: Hardware virtualization**
 - Exceeding constraints lowers performance**
 - Compiler worries about fitting into stripes**
 - NOT about number of stripes**

PipeRench a big VLIW?

- ◆ Stripe configurations are “big instructions”
- ◆ PipeRench instructions keep working
 - More computation per instruction**
 - Better utilization of high “instruction” bandwidth**
- ◆ Similar problems to VLIW:
 - Compilation is challenging
 - Unutilized resources consume “instruction” bandwidth**
 - Wider stripe requires recompilation
 - Forward compatibility from deeper pipe, not wider issue**

PipeRench a Vector Machine?

- ◆ Yes:
 - Pipelining
 - Memory bandwidth requirements
- ◆ No:
 - Interconnect through wires not registers
 - Operations
 - Flexibility

Interfacing

- ◆ Co-processor
- ◆ On I/O:
 - (sensors, speakers, monitors, disks, networks)
- ◆ Internal Functional units
 - Integer, Load/Store, FP, and Reconfigurable
- ◆ In Register File?

Register File Interfacing

◆ Chimera Processor (Northwestern)

- Configurable logic defines functions on registers

- Example:

RFUOP #3 = (R3 == 0) ? 20 : (R1 & R2)

LOAD R1, (R0)+

LOAD R2, (R0)+

LOAD R3, (R0)+

LOAD R4, RFUOP #3

- Uses: complex logical operations, loop conditions

ISP Mindset

- ◆ 40+ years of intellectual investment in ISPs
languages, code, educations, corporations...
- ◆ Can't port existing applications
 - Brute-force
 - Have to go back to initial definition of application
- ◆ My faith:
 - Technical survival-of-the-fittest

Summary

- ◆ Instructions:
Good and bad for media-centric computing
 - Bad: globality, granularity, inflexibility
 - Good: forward-compatibility, compilability
- ◆ FPGA-based computing:
Software without instructions: good or bad
- ◆ PipeRench: Good and not the bad