The Vector-Thread Architecture

Ronny Krashinsky,

Chris Batten, Krste Asanović

Computer Architecture Group

MIT Laboratory for Computer Science

ronny@mit.edu

www.cag.lcs.mit.edu/scale

Boston Area Architecture Workshop (BARC)

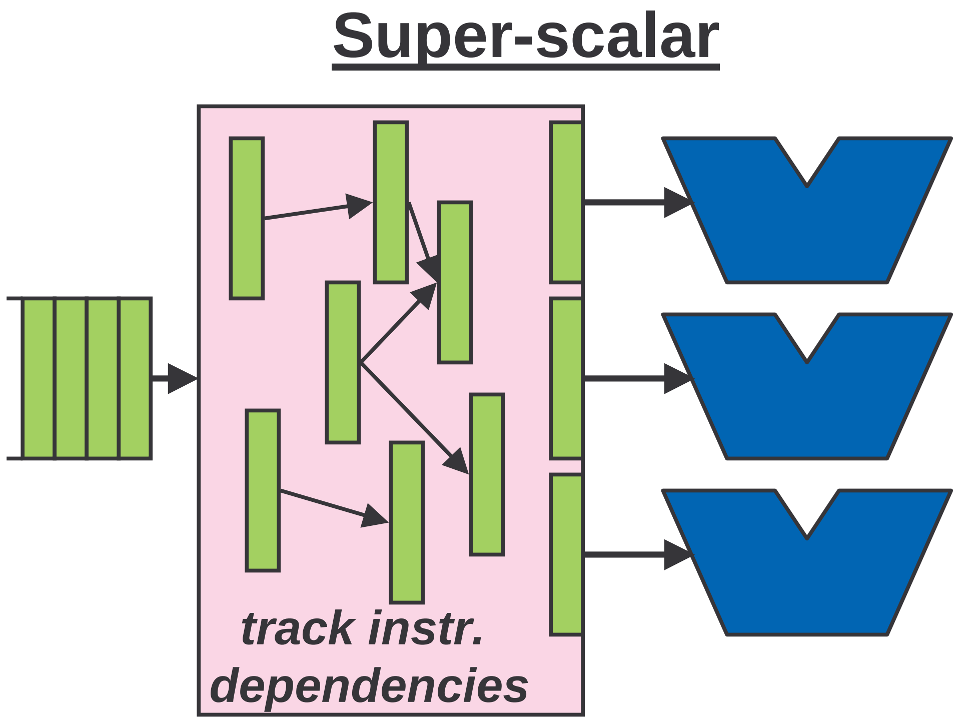
January 30th, 2003

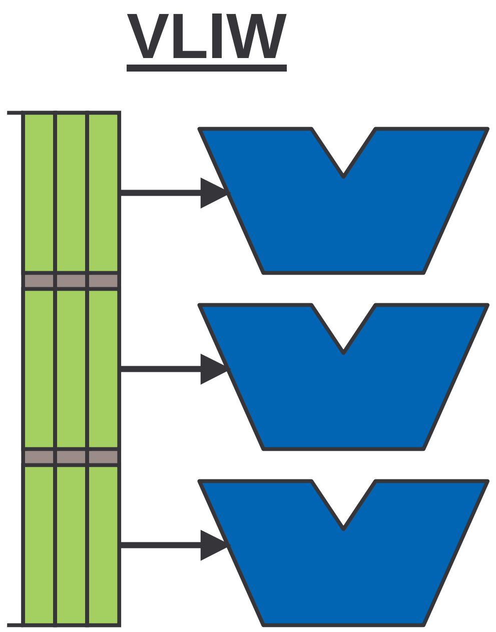
Introduction

- Architectures are all about exploiting the parallelism inherent to applications
  - Performance
  - Energy
- The Vector-Thread Architecture is a new approach which can flexibly take advantage of many forms of parallelism available in different applications
  – instruction, loop, data, thread
- The key goal of the vector-thread architecture is efficiency – high performance with low power consumption and small area
  - A clean, compiler-friendly programming model is key to realizing these goals

Instruction Parallelism

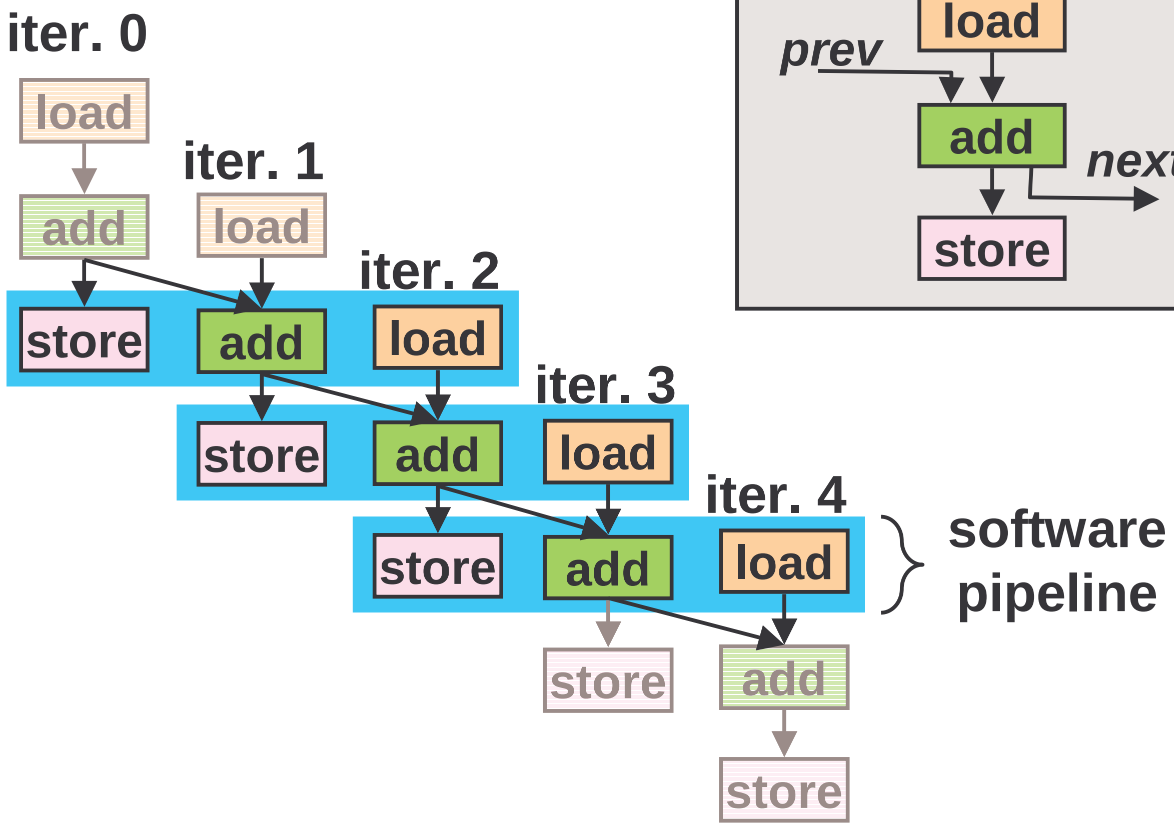
- Independent instructions can execute concurrently
- Super-scalar architectures dynamically schedule instructions in hardware to enable out-of-order and parallel execution
- Software statically schedules parallel instructions on a VLIW machine

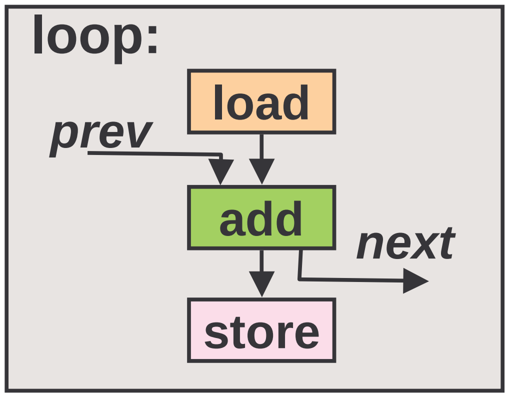


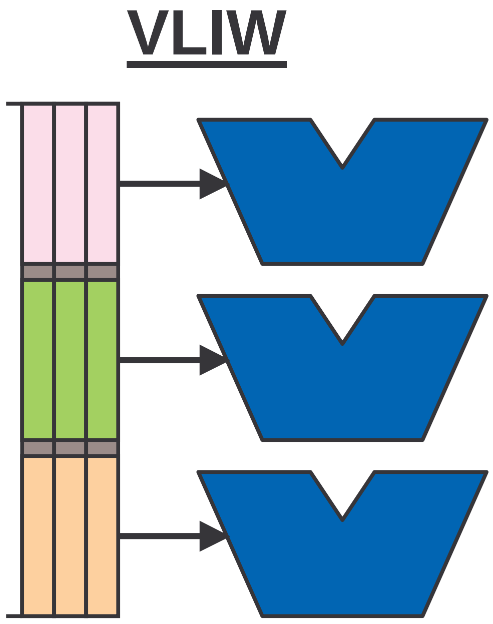


Loop Parallelism

- Operations from disjoint iterations of a loop can execute in parallel
- VLIW architectures use *software pipelining* to statically schedule instructions from different loop iterations to execute concurrently

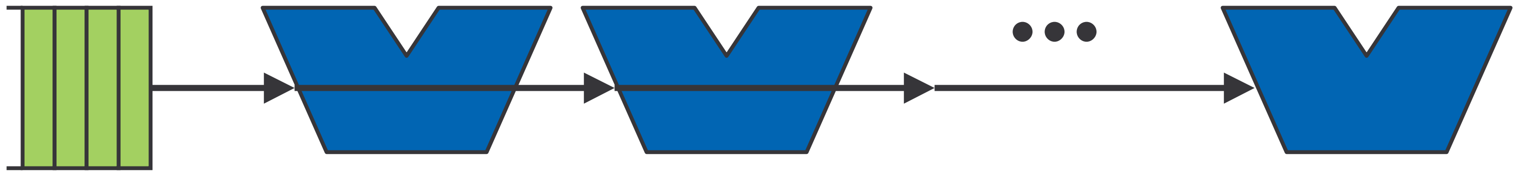






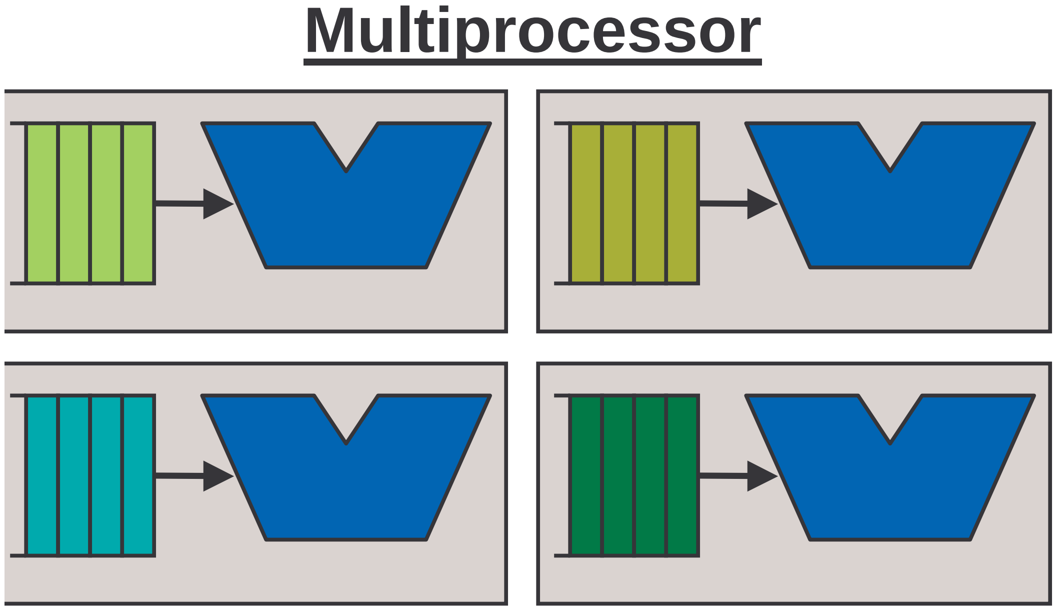
Data Parallelism

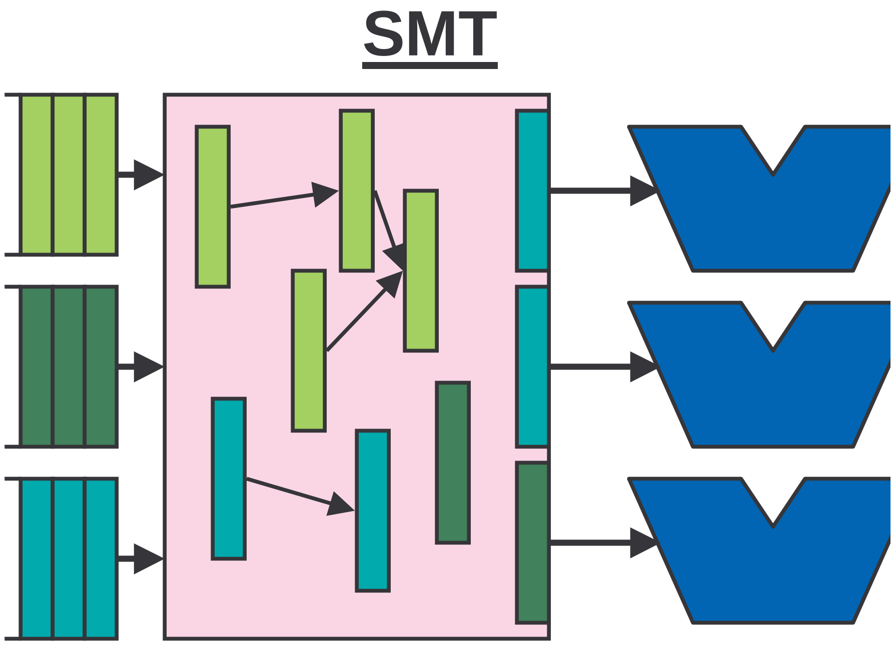
- A single operation can be applied in parallel across a set of data
- In vector architectures, one instruction identifies a set of independent operations which can execute in parallel
- Control overhead can be amortized



Thread Parallelism

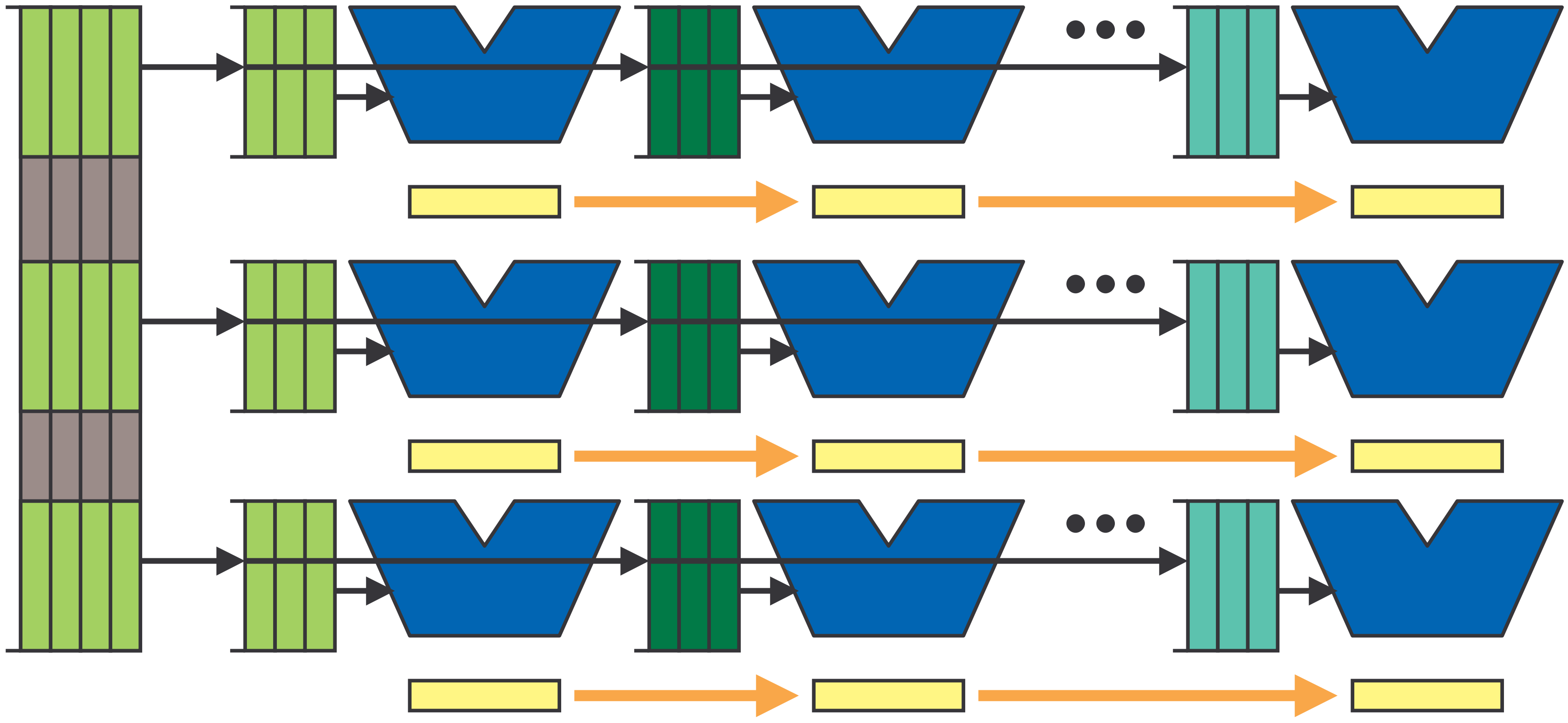
- Separate threads of control can execute concurrently
- Multiprocessor architectures allow different threads to execute at the same time on different processors
- Multithreaded architectures execute multiple threads at the same time to better utilize a single set of processing resources





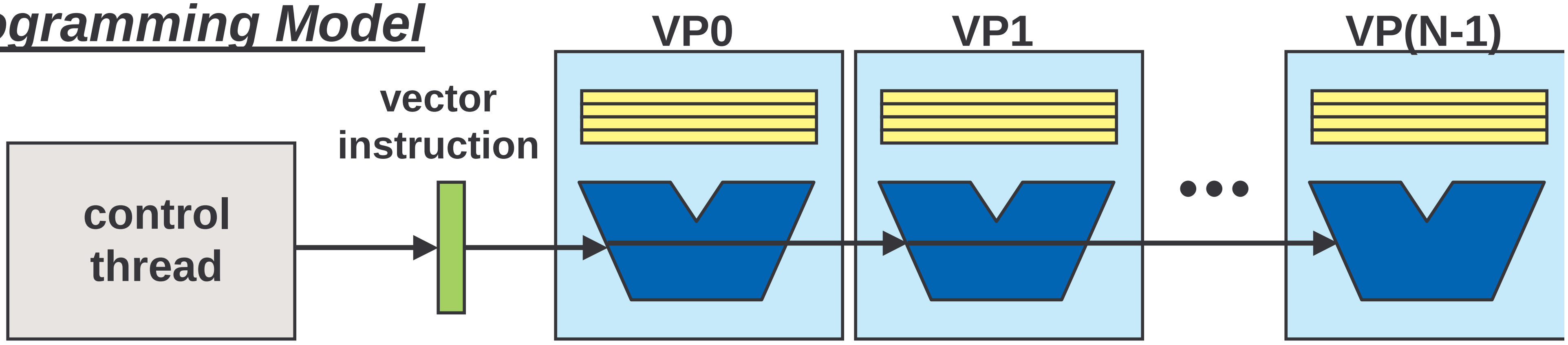
Vector-Thread Architecture Overview

- Data parallelism – start with vector architecture
- Thread parallelism – give execution units local control
- Loop parallelism – allow fine-grain dataflow communication between execution units
- Instruction parallelism – add wide issue



Vector Architecture

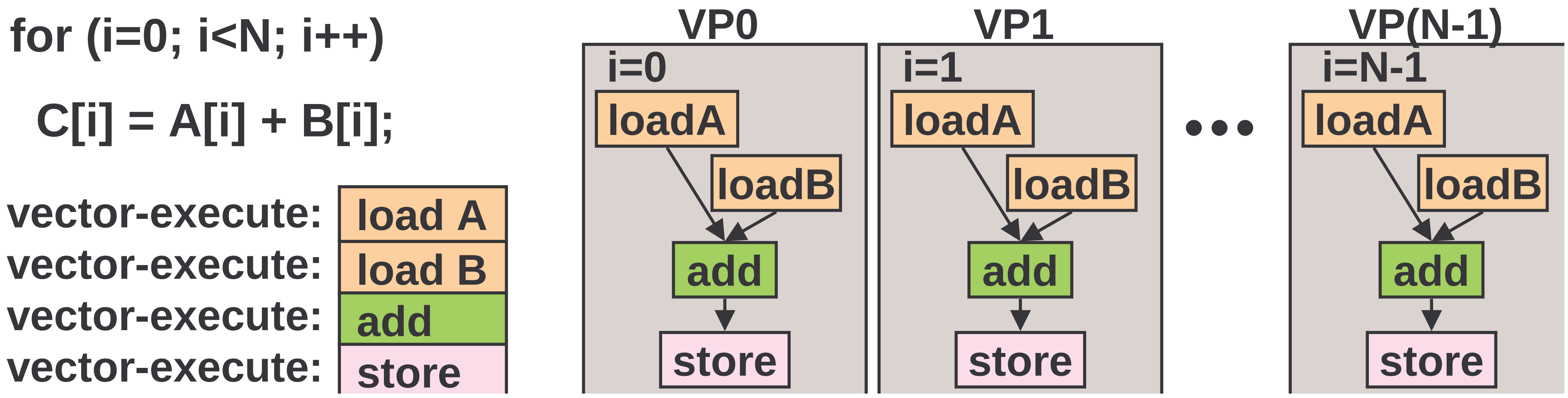
Programming Model



- A *control thread* interacts with a set of *virtual processors* (VPs)
- VPs contain registers and execution units
- VPs execute instructions under *slave control*
- Each iteration in a vectorizable loop mapped to its own VP (w. stripmining)

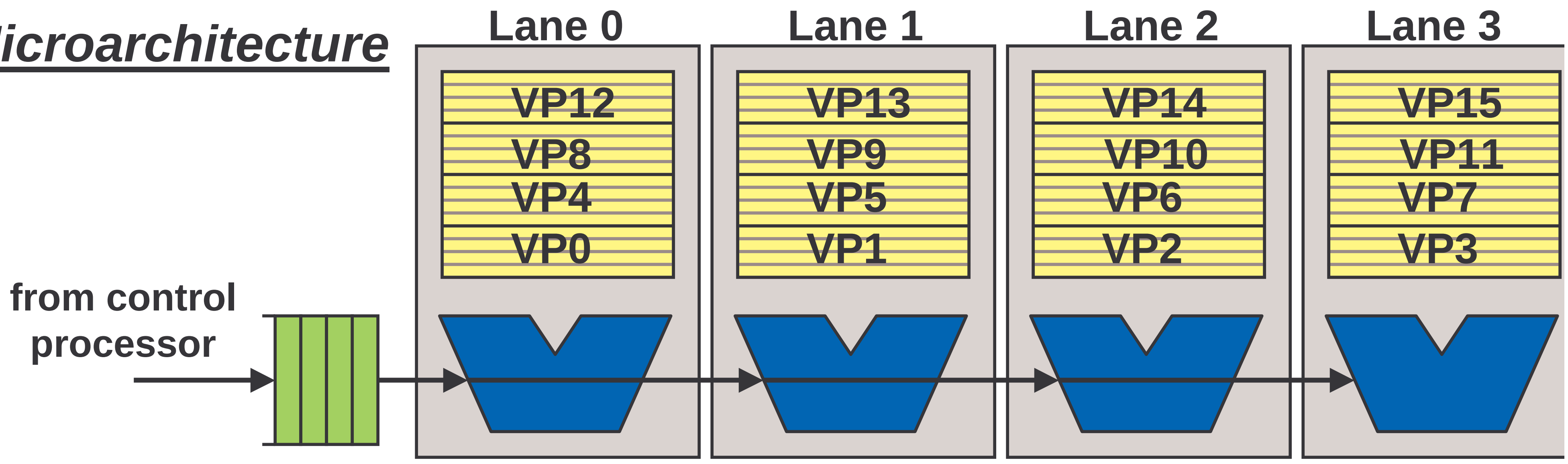
Using VPs for Vectorizable Loops

```
for (i=0; i<N; i++)
  C[i] = A[i] + B[i];
```



Vector Microarchitecture

Microarchitecture



- Lanes contain regfiles and execution units – VPs map to lanes and share physical resources
- Operations execute in parallel across lanes and sequentially for each VP mapped to a lane – control overhead amortized to save energy

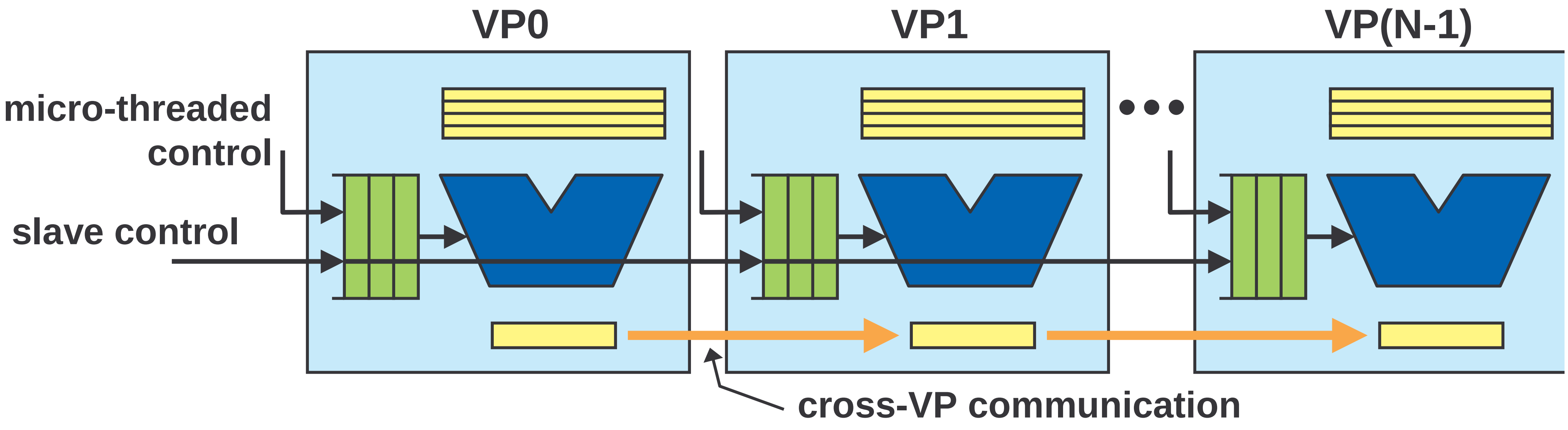
vector-execute:	load A
vector-execute:	load B
vector-execute:	add
vector-execute:	store

Execution on Vector Processo

Lane 0	Lane 1	Lane 2	Lane 3
loadA	loadA	loadA	loadA
loadA	loadA	loadA	loadA
loadA	loadA	loadA	loadA
loadA	loadA	loadA	loadA
loadB	loadB	loadB	loadB
loadB	loadB	loadB	loadB
loadB	loadB	loadB	loadB
loadB	loadB	loadB	loadB
add	add	add	add
add	add	add	add
add	add	add	add
add	add	add	add
store	store	store	store
store	store	store	store
store	store	store	store
store	store	store	store

Vector-Thread Architecture

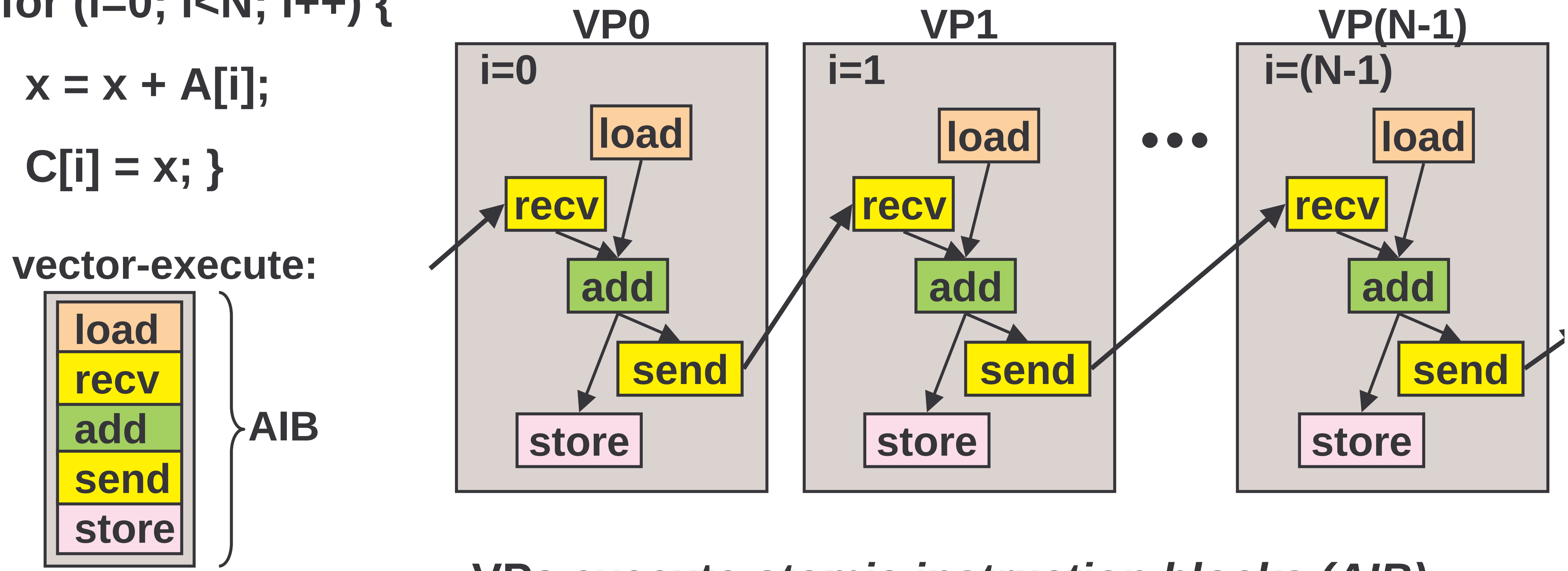
rogramming Model



- Vector of Virtual Processors (similar to traditional vector architecture)
- VPs are decoupled – local instruction queues break the rigid synchronization of vector architectures
- Under *slave control*, the control thread sends instructions to all VPs
- Under *micro-threaded control*, each VP fetches its own instructions
- *Cross-VP communication* allows each VP to send data to its successor

Using VPs for Do-Across Loops

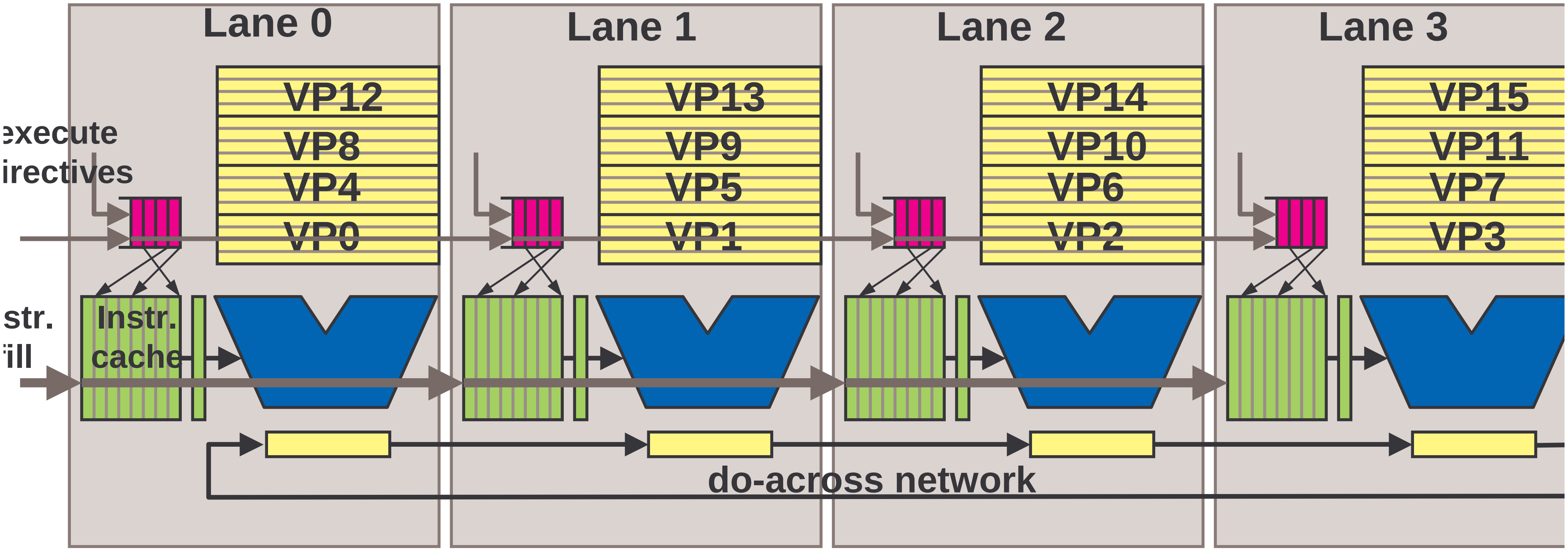
```
for (i=0; i<N; i++) {
  x = x + A[i];
  C[i] = x; }
```



- VPs execute *atomic instruction blocks (AIB)*
- Each iteration in a data dependent loop is mapped to its own VP
- Cross-VP *send* and *recv* operations communicate do-across results from one VP to the next VP (next iteration in time)

Vector-Thread Microarchitecture

Microarchitecture



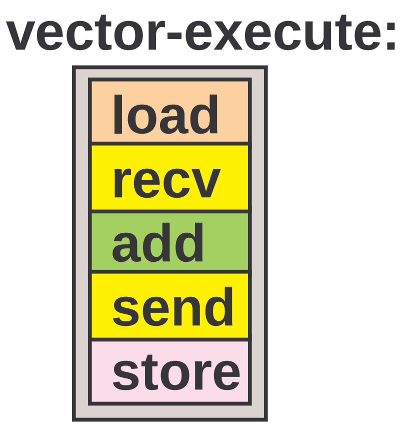
VPs striped across lanes as in traditional vector machine

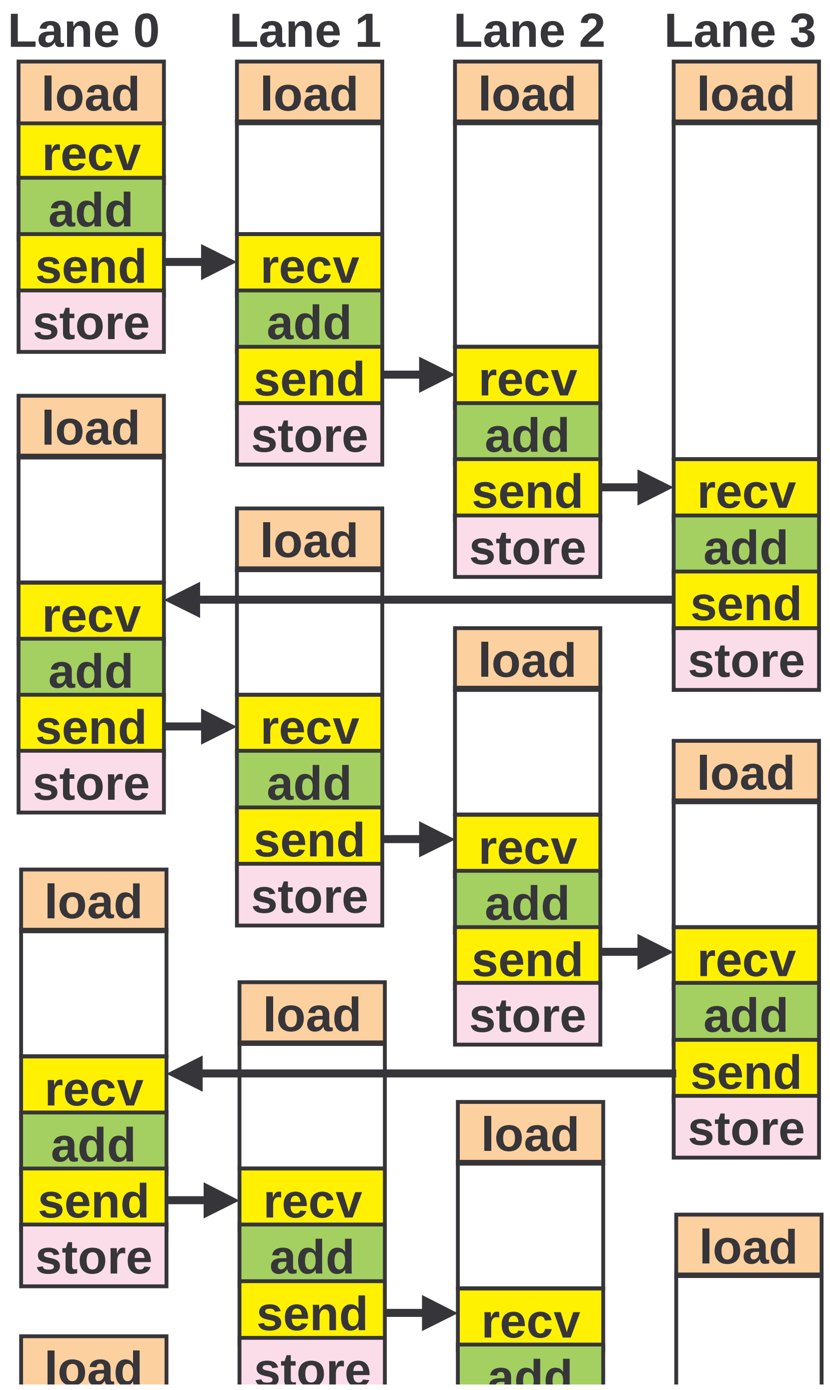
Lanes have small instruction cache (e.g. 32 instr's), decoupled executio

Execute directives point to atomic instruction blocks and indicate which VP(s) the AIB should be executed for – generated by control thread vector-execute command, or VP fetch instruction

Do-across network includes dataflow handshake signals – receiver stall until data is ready

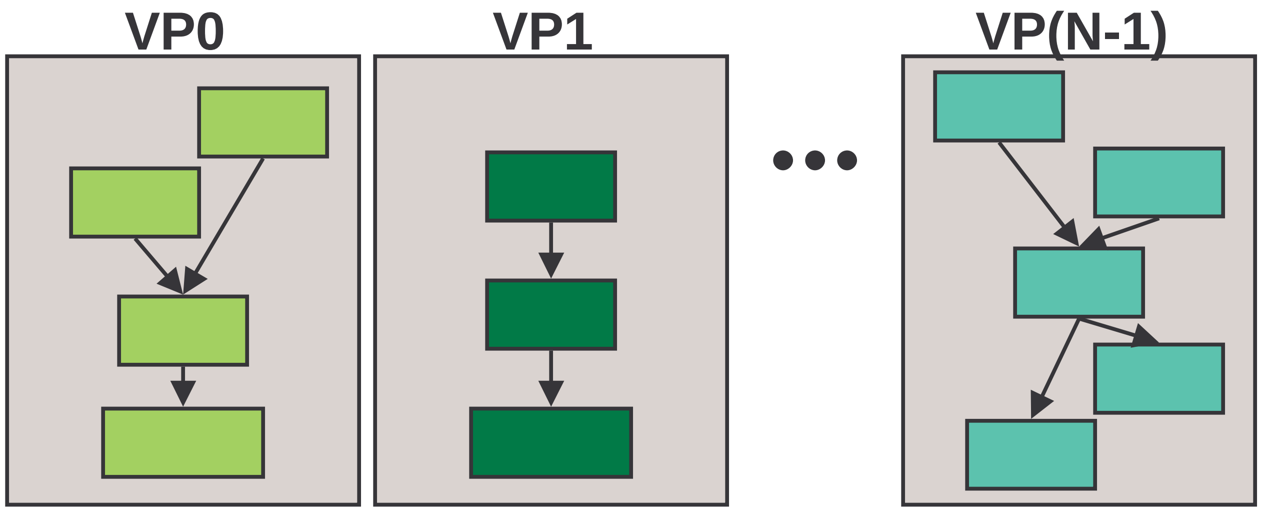
Do-Across Execution





- *Dataflow execution* resolves do-across dependencies dynamically
- Independent instructions execute in parallel – performance adapts to software critical path
- Instruction fetch overhead amortized across loop iterations

Micro-Threading VPs



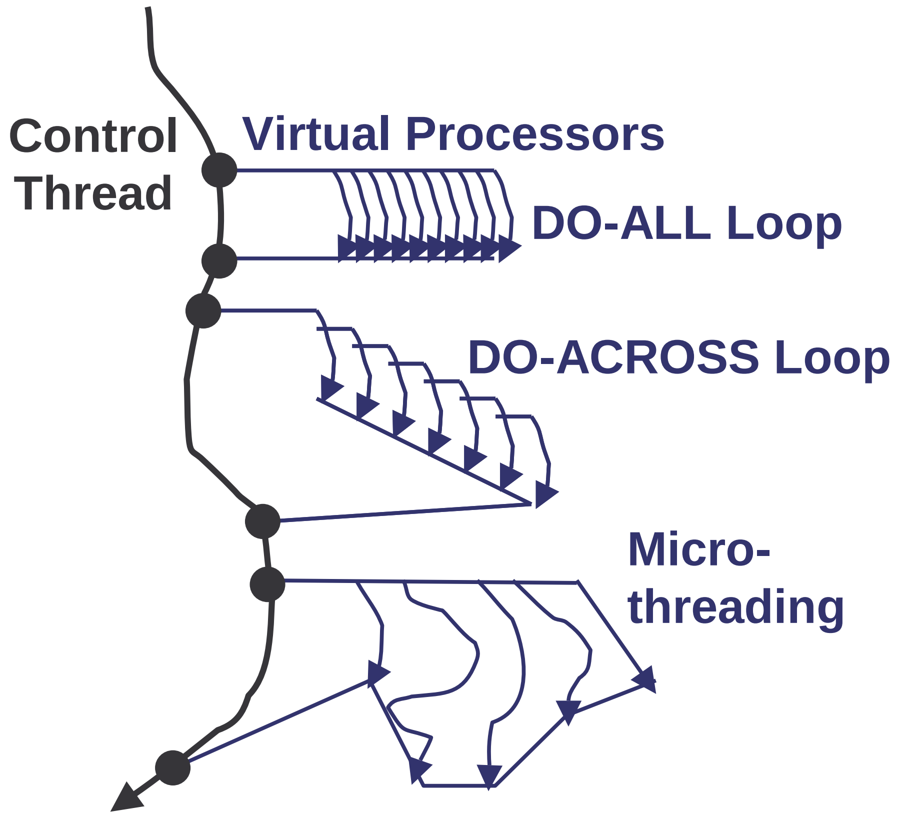
- VPs also have the ability to fetch their own instructions enabling each VP to execute its own thread of control
- Control thread can send a vector fetch instruction to all VPs (i.e. vector fork) – allows efficient thread startup
- Control thread can stall until micro-threads "finish" (stop fetching instructions)
- Enables data-dependent control flow within a loop iteration (alternative to predication)

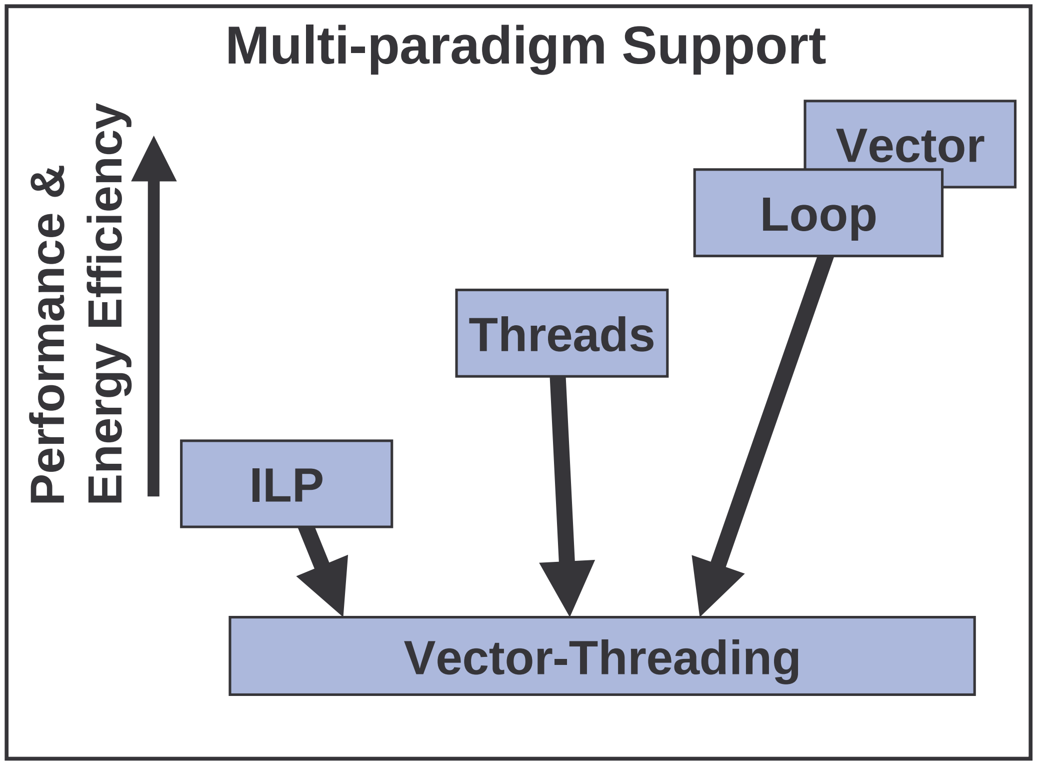
Loop Parallelism and Architectures

Loops are ubiquitous and contain ample parallelism across iterations

- Super-scalar: must track dependencies between all instructions in a loop body (and correctly predict branches) before executing instruction in the subsequent iteration… and do this repeatedly for each loop iteration
- VLIW: software pipelining exposes parallelism, but requires static scheduling which is difficult and inadequate with dynamic latencies and dependencies
- Vector: efficient, but limited to do-all loops, no do-across
- Vector-thread: Software efficiently exposes parallelism, and dynamic dataflow automatically adapts to critical path. Uses simple in-order execution units, and amortizes instruction fetch overhead across loop iterations

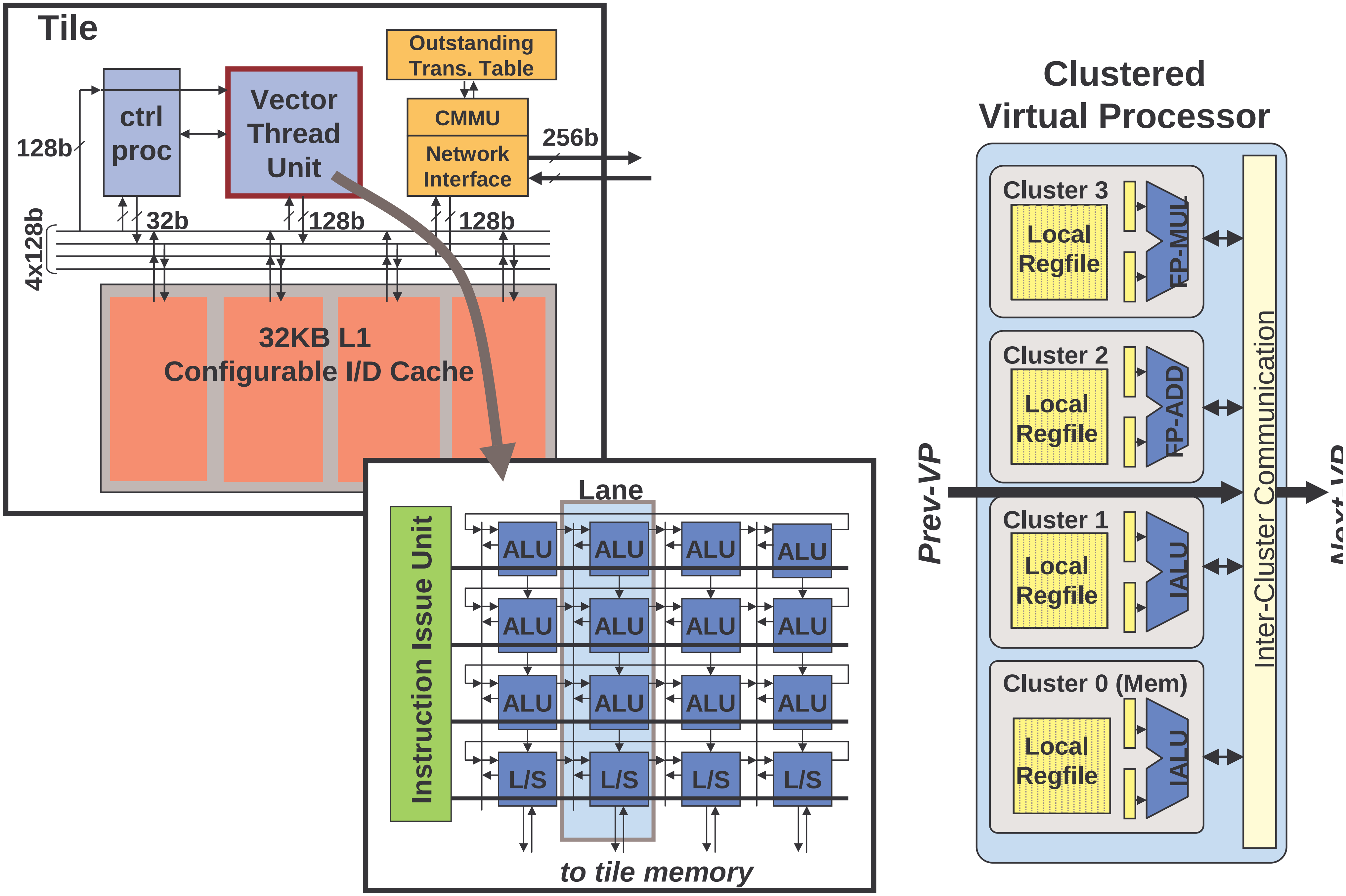
Using the Vector-Thread Architecture





- The Vector-Thread Architecture seeks to efficiently exploit the available parallelism in any given application
- Using the same set of resources, it can flexibly transition from pure data parallel operation, to parallel loop execution with do-across dependencies, to fine-grain multi-threading

SCALE-0 Overview


Tile
ctrl proc
Vector Thread Unit
Outstanding Trans. Table
CMMU
Network Interface
256b
128b
32b
128b
128b
4x128b
32KB L1 Configurable I/D Cache
Lane
Instruction Issue Unit
ALU
L/S
to tile memory
Clustered Virtual Processor
Cluster 3
Local Regfile
FP-MUL
Cluster 2
FP-ADD
Cluster 1
IALU
Cluster 0 (Mem)
Inter-Cluster Communication
Prev-VP
Next-VP