

Chapter 4

Instruction

Level

Parallelism

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
 - Deeper pipeline
 - Less work per stage \Rightarrow shorter clock cycle
 - Multiple issue
 - Replicate pipeline stages \Rightarrow multiple pipelines
 - Start multiple instructions per clock cycle
 - $CPI < 1$, so use Instructions Per Cycle (IPC)
 - E.g., 4GHz 4-way multiple-issue
 - 16 BIPS, peak $CPI = 0.25$, peak $IPC = 4$
 - But dependencies reduce this in practice

Multiple Issue

- Fetch multiple instructions (e.g. 2, 4) in parallel, and then consider how many and which of them to execute in parallel
- Static multiple issue
 - Compiler groups instructions to be issued together
 - Packages them into “issue slots”
 - Compiler detects and avoids hazards, where none independent available, fills-in noop’s
- Dynamic multiple issue
 - CPU examines instruction stream and chooses instructions to issue each cycle
 - Compiler can help by reordering instructions
 - CPU resolves hazards using advanced techniques at runtime

Speculation

- “Guess” what to do with an instruction
 - Start operation as soon as possible
 - Check whether guess was right
 - If so, complete the operation
 - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
 - Speculate on branch outcome
 - Roll back if path taken is different
 - Speculate on load
 - Roll back if location is updated

Compiler/Hardware Speculation

- Compiler can reorder instructions
 - e.g., move load before branch
 - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
 - Buffer results until it determines they are actually needed
 - Flush buffers on incorrect speculation

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
 - e.g., speculative load before null-pointer check
- Static speculation
 - Can add ISA support for deferring exceptions
- Dynamic speculation
 - Can buffer exceptions until instruction completion (which may not occur)

Static Multiple Issue

- Compiler groups instructions into “issue packets”
 - Group of instructions that can be issued on a single cycle
 - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
 - Specifies multiple concurrent operations
 - ⇒ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

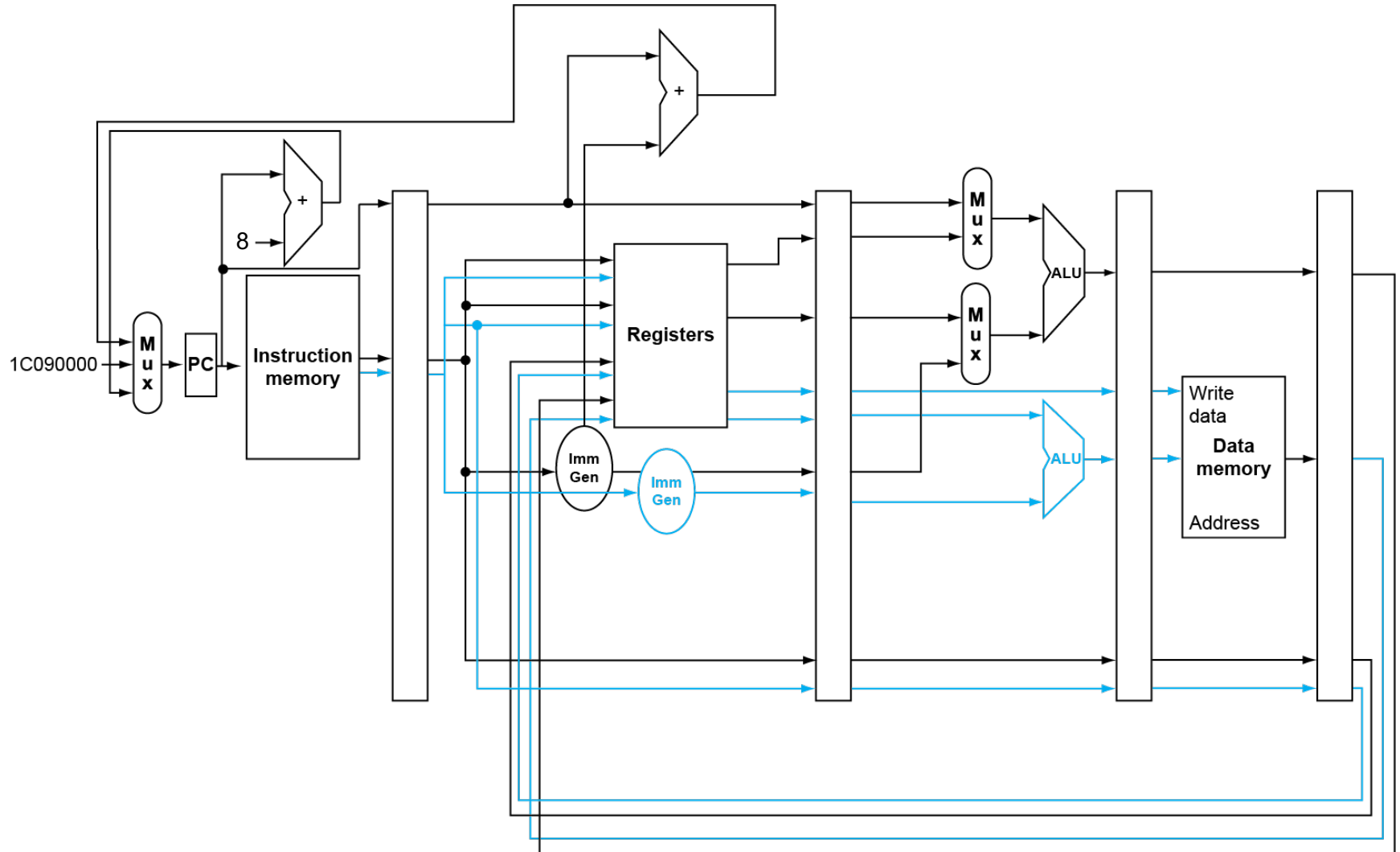
- Compiler must remove some/all hazards
 - Reorder instructions into issue packets
 - No dependencies within a packet
 - Possibly some dependencies between packets
 - Varies between ISAs; compiler must know!
 - Pad with nop if necessary

RISC-V with Static Dual Issue

- Two-issue packets
 - One ALU/branch instruction
 - One load/store instruction
 - 64-bit aligned
 - ALU/branch, then load/store
 - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

RISC-V with Static Dual Issue



Hazards in the Dual-Issue RISC-V

- More instructions executing in parallel
- EX data hazard
 - Forwarding avoided stalls with single-issue
 - Now can't use ALU result in load/store in same packet
 - `add x10, x0, x1`
`ld x2, 0(x10)`
 - Split into two packets, effectively a stall
- Load-use hazard
 - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Scheduling Example

- Schedule this for dual-issue RISC-V

```
Loop: 1d    x31,0(x20)    // x31=array element
      add   x31,x31,x21   // add scalar in x21
      sd    x31,0(x20)   // store result
      addi  x20,x20,-8    // decrement pointer
      blt   x22,x20,Loop // branch if x22 < x20
```

	ALU/branch	Load/store	cycle
Loop:	nop	1d x31,0(x20)	1
	addi x20,x20,-8	nop	2
	add x31,x31,x21	nop	3
	blt x22,x20,Loop	sd x31,8(x20)	4

- $IPC = 5/4 = 1.25$ (c.f. peak $IPC = 2$)

Loop Unrolling

- Replicate loop body to expose more parallelism
 - Reduces loop-control overhead
- Use different registers per replication
 - Called “register renaming”
 - Avoid loop-carried “anti-dependencies”
 - Store followed by a load of the same register
 - Aka “name dependence”
 - Reuse of a register name

Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi x20,x20,-32	ld x28, 0(x20)	1
	nop	ld x29, 24(x20)	2
	add x28,x28,x21	ld x30, 16(x20)	3
	add x29,x29,x21	ld x31, 8(x20)	4
	add x30,x30,x21	sd x28, 32(x20)	5
	add x31,x31,x21	sd x29, 24(x20)	6
	nop	sd x30, 16(x20)	7
	b1t x22,x20,Loop	sd x31, 8(x20)	8

- $IPC = 14/8 = 1.75$
 - Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, ... each cycle
 - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
 - Though it may still help
 - Code semantics ensured by the CPU
 - Allows executables to run on newer processors, with same ISA but different pipeline, without needing to be recompiled

Dynamic Pipeline Scheduling

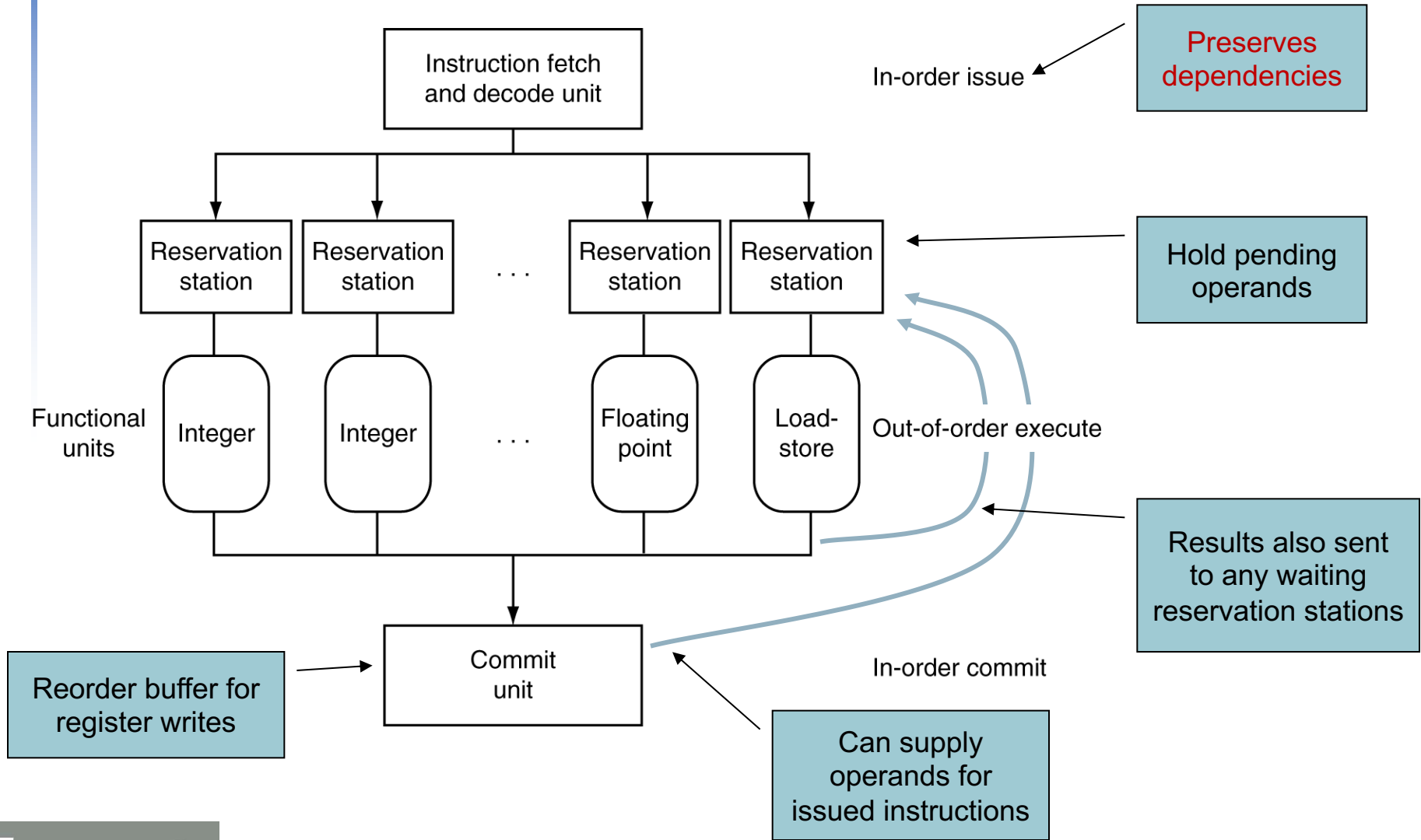
- Allow the CPU to execute instructions out of order to avoid stalls
 - But commit result to registers in order
- Out-of-Order (OoO) Execution (RAW, WAR, WAW)
 - In-Order Commit (can flush results of mis-speculated instructions, and also allow precise exceptions)

- Example

```
ld    x31, 20(x21)
add   x1, x31, x2
sub   x23, x23, x3
andi  x5, x23, 20
```

- Can start sub while add is waiting for ld

Dynamically Scheduled CPU



Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
 - If operand is available in register file or reorder buffer
 - Copied to reservation station
 - No longer required in the register; can be overwritten
 - If operand is not yet available
 - It will be provided to the reservation station by a function unit
 - Register update may not be required

Speculation

- Predict branch and continue issuing
 - Don't commit until branch outcome determined
- Load speculation
 - Avoid load and cache miss delay
 - Predict the effective address
 - Predict loaded value
 - Load before completing outstanding stores
 - Bypass stored values to load unit
 - Don't commit load until speculation cleared

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
 - e.g., [cache misses](#)
- Can't always schedule around branches
 - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
 - e.g., pointer aliasing
- Some parallelism is hard to expose
 - Limited window size during instruction issue
- Memory delays and limited bandwidth
 - Hard to keep pipelines full
- Speculation can help if done well

Chapter 6

Parallelism

SIMD, Vector, GPU

Pitfall: Amdahl's Law

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

e.g. available parallelism → improvement factor
 e.g. parallelizable → T_{affected}
 e.g. non-parallelizable → $T_{\text{unaffected}}$

- E.g.: Multiply accounts for 80s out of total 100s
 - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20 \quad \blacksquare \text{ Can't be done!}$$

- Corollary: make the common case fast

Scaling Example

- Workload: sum of 10 scalars, and 10×10 matrix sum
 - Speed up from 10 to 100 processors
- Single processor: Time = $(10 + 100) \times t_{\text{add}}$
- 10 processors
 - Time = $10 \times t_{\text{add}} + 100/10 \times t_{\text{add}} = 20 \times t_{\text{add}}$
 - Speedup = $110/20 = 5.5$ (55% of potential)
- 100 processors
 - Time = $10 \times t_{\text{add}} + 100/100 \times t_{\text{add}} = 11 \times t_{\text{add}}$
 - Speedup = $110/11 = 10$ (10% of potential)
- Assumes load can be balanced across processors

Scaling Example (cont)

- What if matrix size is 100×100 ?
- Single processor: Time = $(10 + 10000) \times t_{\text{add}}$
- 10 processors
 - Time = $10 \times t_{\text{add}} + 10000/10 \times t_{\text{add}} = 1010 \times t_{\text{add}}$
 - Speedup = $10010/1010 = 9.9$ (99% of potential)
- 100 processors
 - Time = $10 \times t_{\text{add}} + 10000/100 \times t_{\text{add}} = 110 \times t_{\text{add}}$
 - Speedup = $10010/110 = 91$ (91% of potential)
- Assuming load balanced

Strong vs Weak Scaling

- Strong scaling: problem size fixed
 - As in example
- Weak scaling: problem size proportional to number of processors
 - 10 processors, 10×10 matrix
 - Time = $20 \times t_{\text{add}}$
 - 100 processors, 32×32 matrix
 - Time = $10 \times t_{\text{add}} + 1000/100 \times t_{\text{add}} = 20 \times t_{\text{add}}$
 - Constant performance in this example

Instruction and Data Streams

- An alternate classification

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE, AVX instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345

- SPMD: Single Program Multiple Data
 - A parallel program on a MIMD computer
 - Conditional code for different processors

SIMD

- Operate elementwise on vectors of data
 - E.g., MMX, SSE, AVX instructions in x86
 - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
 - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

Vector Processors

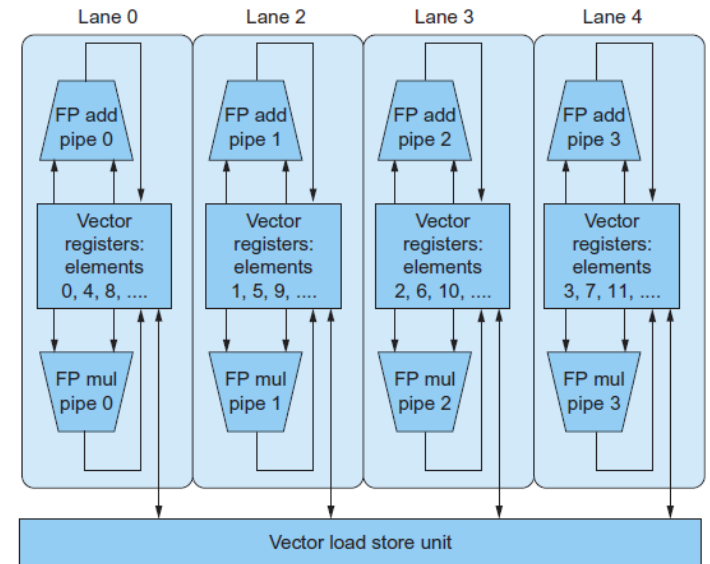
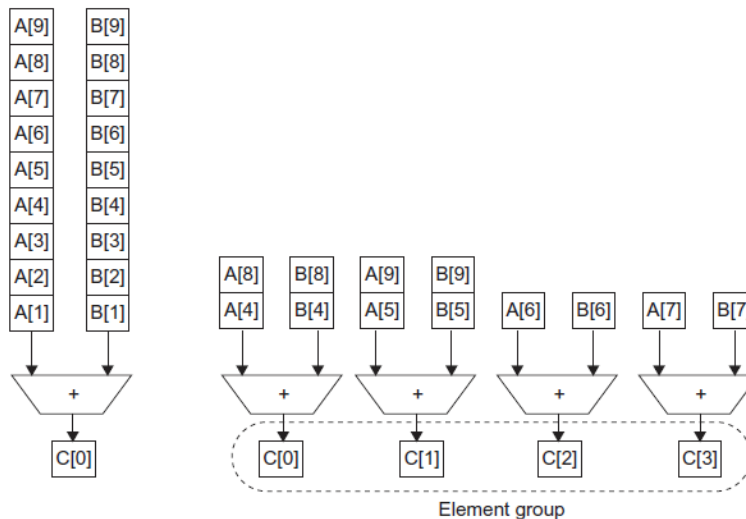
- Highly pipelined function units
- Stream data from/to vector registers to units
 - Data collected from memory into registers
 - Results stored from registers to memory
- Example: Vector extension to RISC-V
 - v0 to v31: 32×64 -element registers, (64-bit elements)
 - Vector instructions
 - f1d.v, fsd.v: load/store vector
 - fadd.d.v: add vectors of double
 - fadd.d.vs: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Vector vs. Scalar

- Vector architectures and compilers
 - Simplify data-parallel programming
 - Explicit statement of absence of loop-carried dependences
 - Reduced checking in hardware
 - Regular access patterns benefit from interleaved and burst memory
 - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE, AVX)
 - Better match with compiler technology

Vector vs. Multimedia Extensions

- Vector instructions have variable vector length/width, multimedia extensions have fixed
- Vector instructions support strided access, multimedia extensions do not
- Vector units can be combination of pipelined and arrayed functional units:



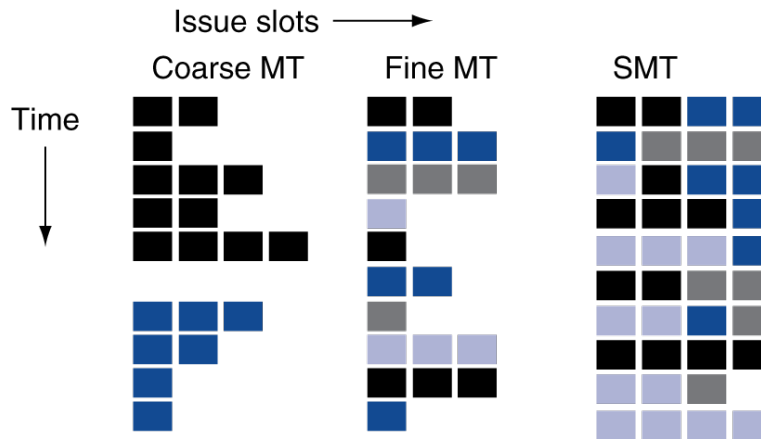
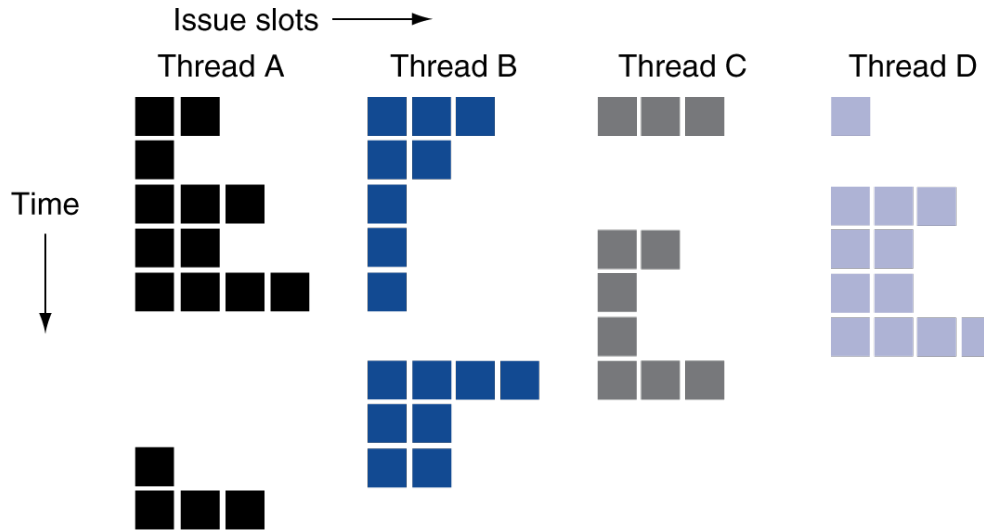
Multithreading

- Performing multiple threads of execution in parallel
 - Replicate registers, PC, etc.
 - Fast switching between threads
- Fine-grain multithreading
 - Switch threads after each cycle
 - Interleave instruction execution
 - If one thread stalls, others are executed
- Coarse-grain multithreading
 - Only switch on long stall (e.g., L2-cache miss)
 - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
 - Schedule instructions from multiple threads
 - Instructions from independent threads execute when function units are available
 - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
 - Two threads: duplicated registers, shared function units and caches

Multithreading Example

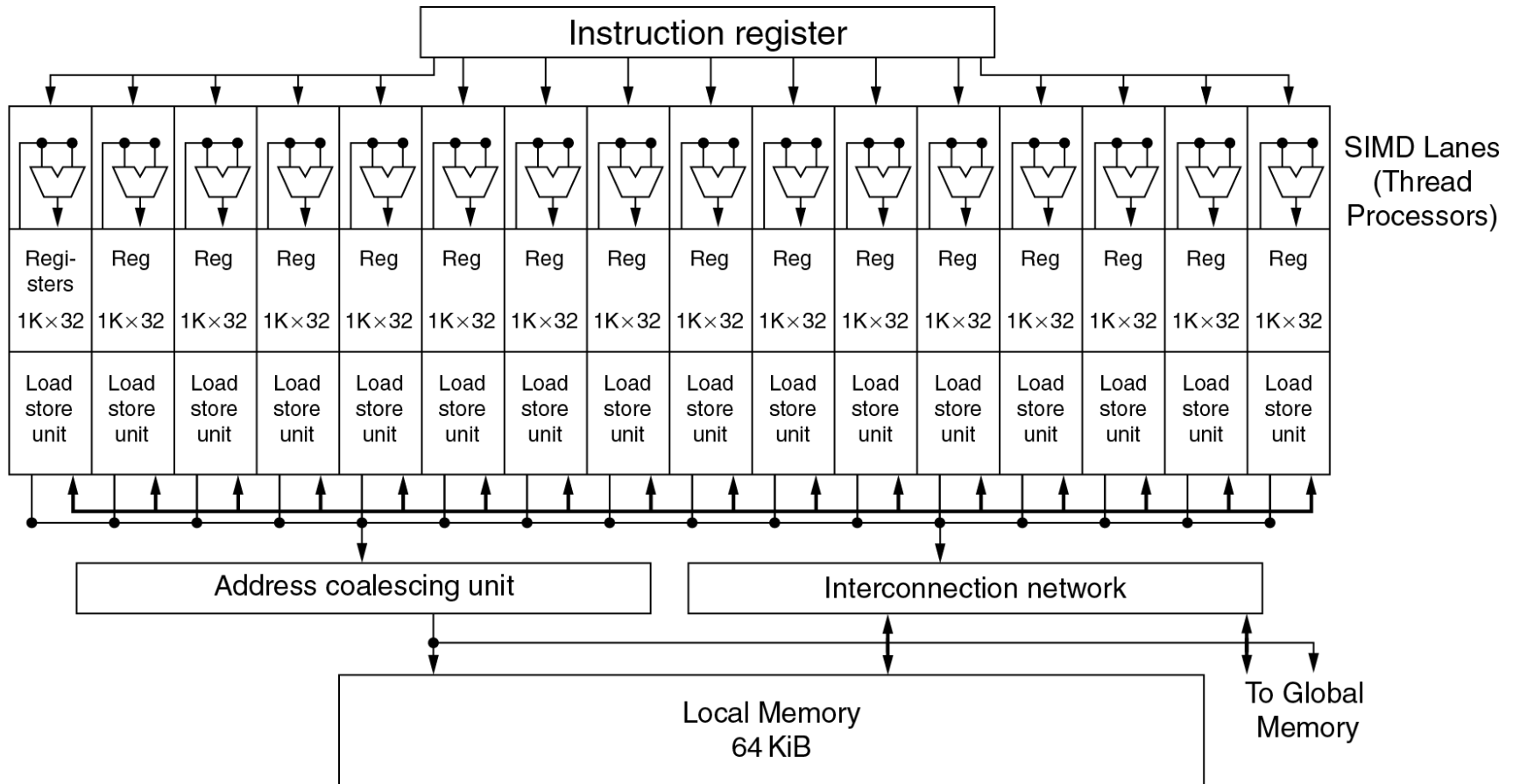


GPU Architectures

- Processing is highly data-parallel
 - GPUs are highly multithreaded
 - Use thread switching to hide memory latency
 - Less reliance on multi-level caches
 - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
 - Heterogeneous CPU/GPU systems
 - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
 - DirectX, OpenGL
 - C for Graphics (Cg), High Level Shader Language (HLSL)
 - Compute Unified Device Architecture (CUDA)

Example: NVIDIA Fermi

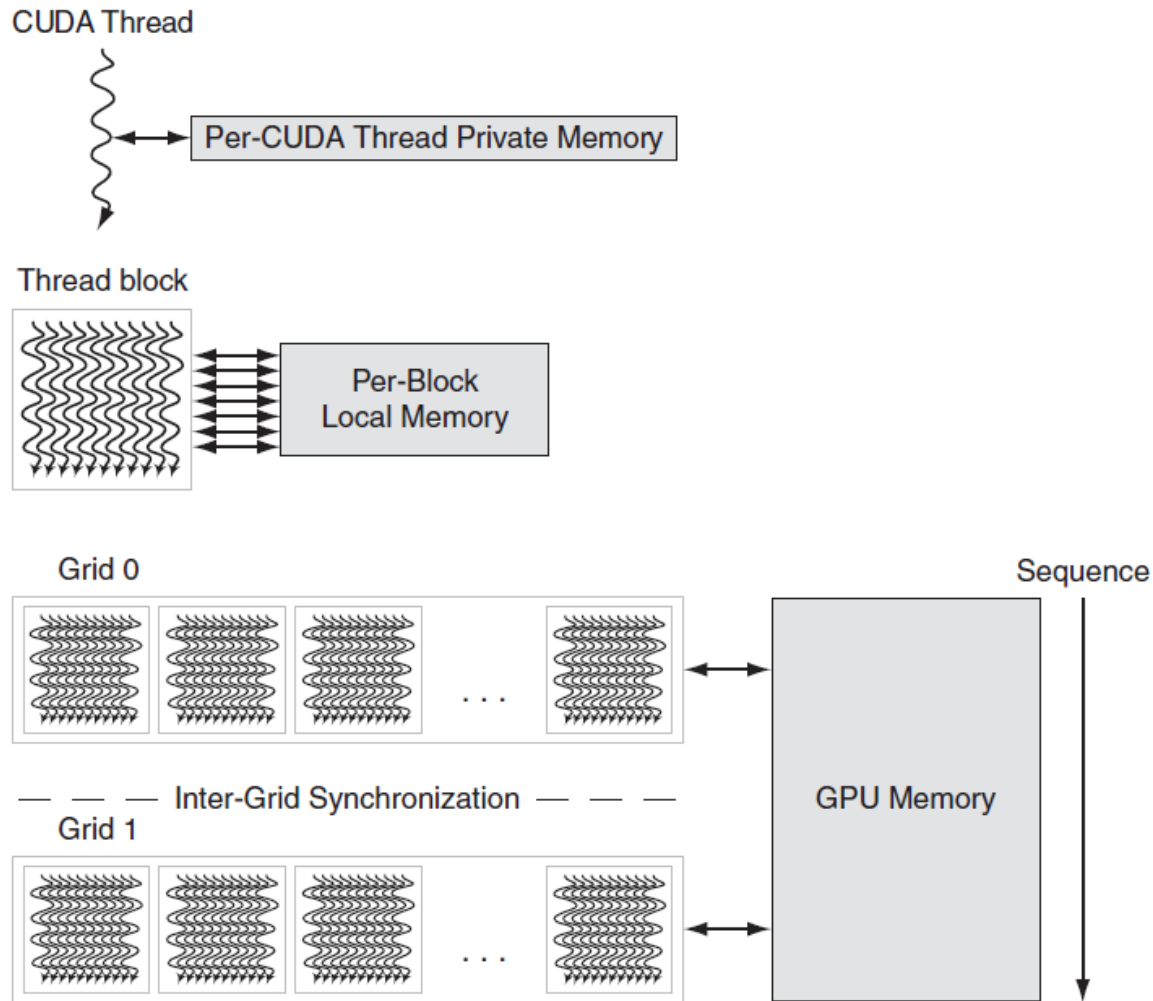
- Multiple SIMD processors, each as shown:



Example: NVIDIA Fermi

- SIMD Processor: 16 SIMD lanes
- SIMD instruction
 - Operates on 32 element wide threads
 - Dynamically scheduled on 16-wide processor over 2 cycles
- 32K x 32-bit registers spread across lanes
 - 64 registers per thread context

GPU Memory Structures



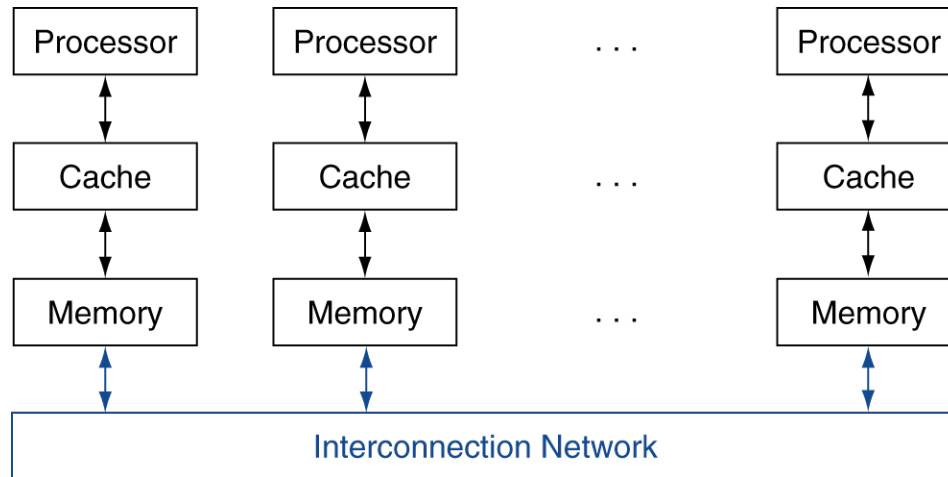
Classifying GPUs

- Don't fit nicely into SIMD/MIMD model
 - Conditional execution in a thread allows an illusion of MIMD
 - But with performance degradation
 - Need to write general purpose code with care

	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	Tesla Multiprocessor

Message Passing

- Each processor has private physical address space
- Hardware sends/receives messages between processors



Loosely Coupled Clusters

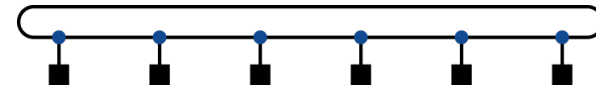
- Network of independent computers
 - Each has private memory and OS
 - Connected using I/O system
 - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
 - Web servers, databases, simulations, ...
- High availability, scalable, affordable
- Problems
 - Administration cost (prefer virtual machines)
 - Low interconnect bandwidth
 - c.f. processor/memory bandwidth on an SMP

Interconnection Networks

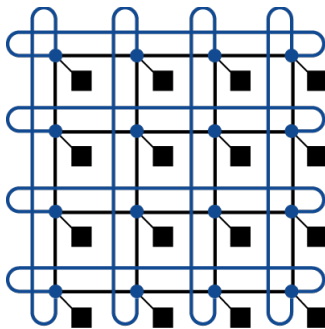
- Network topologies
 - Arrangements of processors, switches, and links



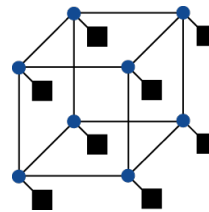
Bus



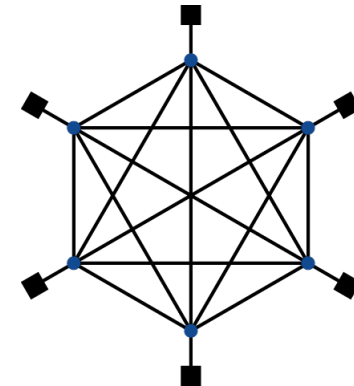
Ring



2D Mesh



N-cube (N = 3)



Fully connected