CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 09 – Multithreading/Vector/GPU

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 09-1 – Multithreading

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

Thread-Level Parallelism (TLP)

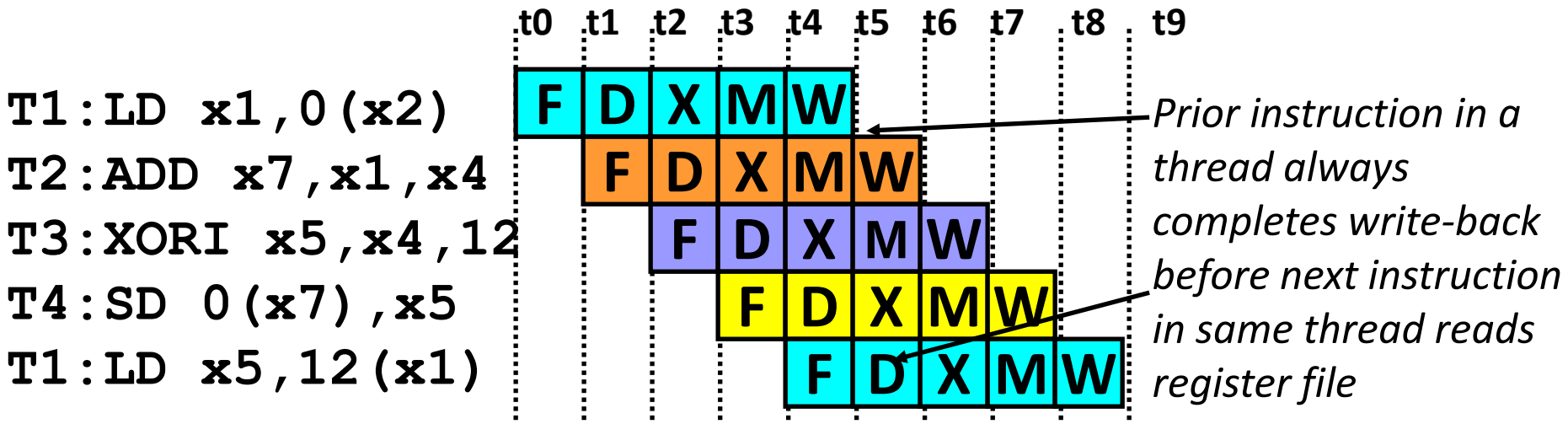
- Difficult to continue to extract instruction-level parallelism (ILP) from a single sequential thread of control
- Many workloads can make use of thread-level parallelism:
  - TLP from ***multiprogramming*** (run independent sequential jobs)
  - TLP from ***multithreaded*** applications (run one job faster using parallel threads)
- Multithreading uses TLP to improve utilization of a single processor

Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

One way is to interleave execution of instructions from different program threads on same pipeline

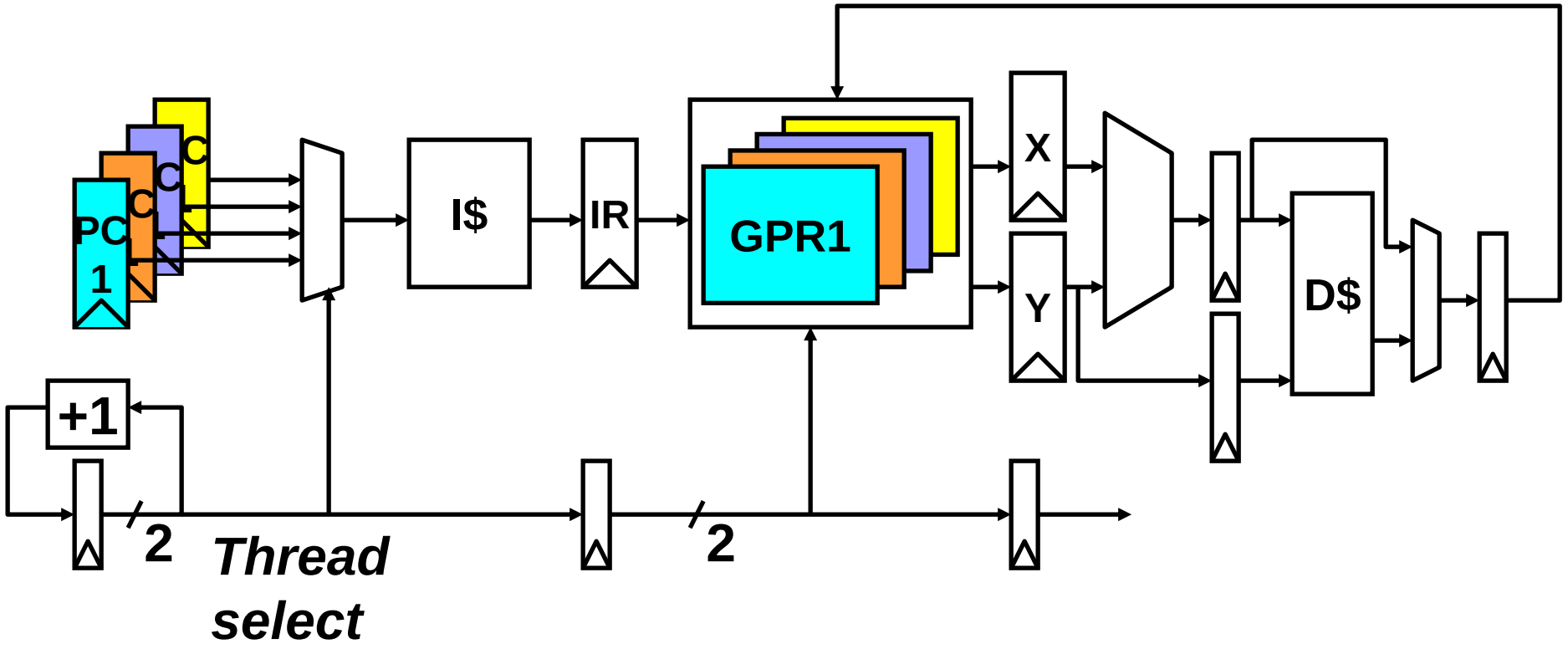
*Interleave 4 threads, T1-T4, on **non-bypassed** 5-stage pipe*



CDC 6600 Peripheral Processors
(Cray, 1964)

- First multithreaded hardware
- 10 “virtual” I/O processors
- Fixed interleave on simple pipeline
- Pipeline has 100ns cycle time
- Each virtual processor executes one instruction every 1000ns
- Accumulator-based instruction set to reduce processor state

Simple Multithreaded Pipeline



- Have to carry thread select down pipeline to ensure correct state bits read/written at each pipe stage
- Appears to software (including OS) as multiple, albeit slower, CPUs

Multithreading Costs

- Each thread requires its own user state
  - PC
  - GPRs
- Also, needs its own system state
  - Virtual-memory page-table-base register
  - Exception-handling registers
- *Other overheads:*
  - Additional cache/TLB conflicts from competing threads
    - or add larger cache/TLB capacity
  - More OS overhead to schedule more threads (where do all these threads come from?)

Thread Scheduling Policies

- Fixed interleave *(CDC 6600 PPUs, 1964)*
  - Each of N threads executes one instruction every N cycles
  - If thread not ready to go in its slot, insert pipeline bubble

- Software-controlled interleave *(TI ASC PPUs, 1971)*
  - OS allocates S pipeline slots amongst N threads
  - Hardware performs fixed interleave over S slots, executing whichever thread is in that slot

- Hardware-controlled thread scheduling *(HEP, 1982)*
  - Hardware keeps track of which threads are ready to go
  - Picks next thread to execute based on hardware priority scheme

Denelcor HEP
(Burton Smith, 1982)

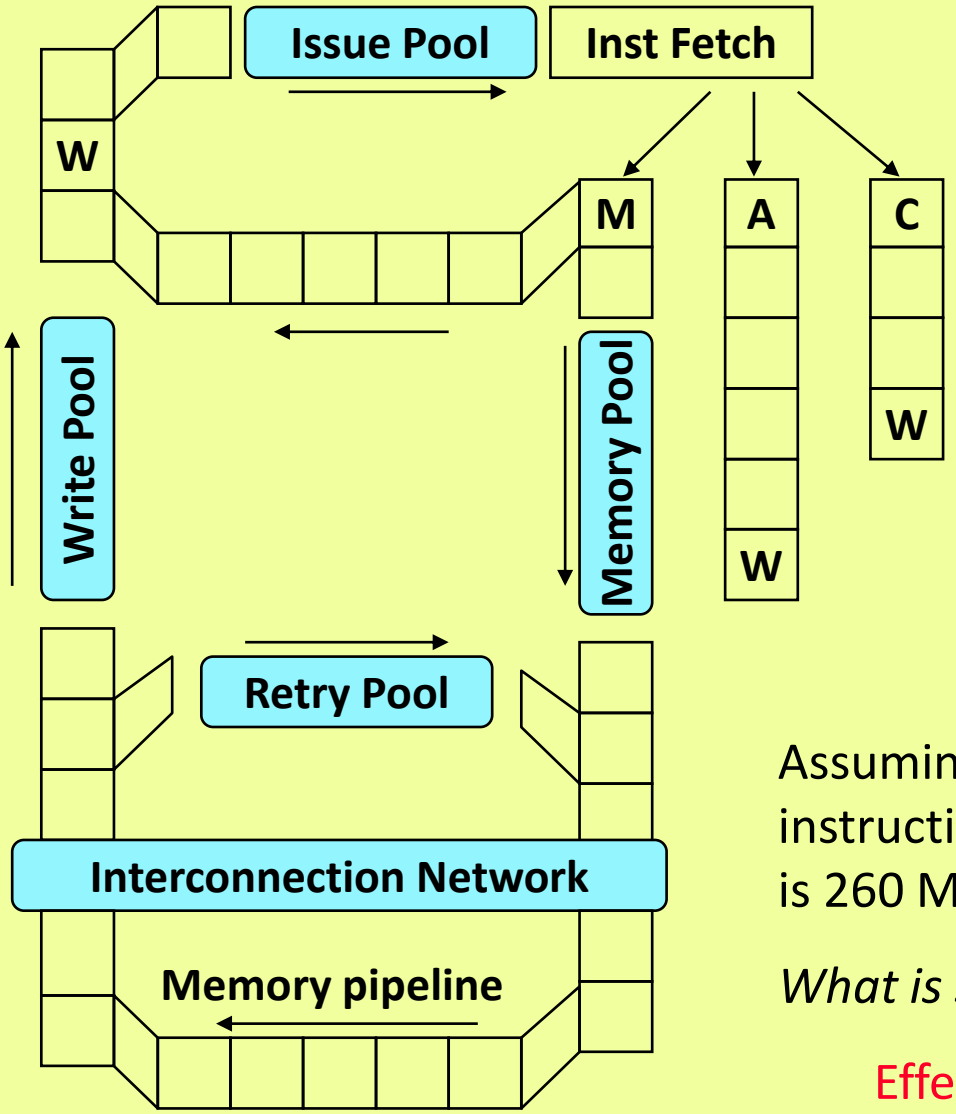
First commercial machine to use hardware threading in main CPU

- 120 threads per processor
- 10 MHz clock rate
- Up to 8 processors
- precursor to Tera MTA (Multithreaded Architecture)

Tera/Cray MTA (1990-2015)

- Up to 256 processors
- Up to 128 active threads per processor
- Processors and memory modules populate a sparse 3D torus interconnection fabric
- Flat, shared main memory
  - No data cache
  - Sustains one main memory access per cycle per processor
- GaAs logic in prototype, 1KW/processor @ 260MHz
  - Second version CMOS, MTA-2, 50W/processor
  - Newer version, XMT, fits into AMD Opteron socket, runs at 500MHz
  - Newer versions (2011), XMT-2, has higher memory bandwidth and capacity
  - Discontinued 2015?

MTA Pipeline



- Every cycle, one VLIW instruction from one active thread is launched into pipeline
- Instruction pipeline is 21 cycles long
- Memory operations incur ~150 cycles of latency

Assuming a single thread issues one instruction every 21 cycles, and clock rate is 260 MHz…

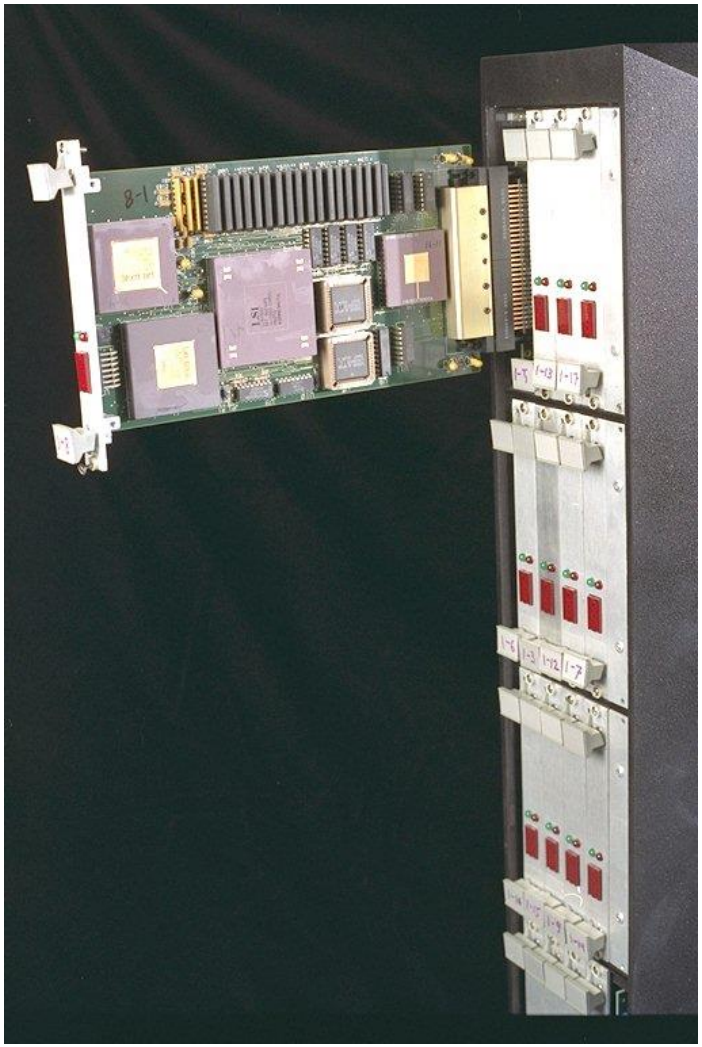
What is single-thread performance?

Effective single-thread issue rate is 260/21 = 12.4 MIPS

Coarse-Grain Multithreading

- Tera MTA designed for supercomputing applications with large data sets and low locality
  - No data cache
  - Many parallel threads needed to hide large memory latency
- Other applications are more cache friendly
  - Few pipeline bubbles if cache mostly has hits
  - Just add a few threads to hide occasional cache miss latencies
  - Swap threads on cache misses

MIT Alewife (1990)



- Modified SPARC chips
  - register windows hold different thread contexts
- Up to four threads per node
- Thread switch on local cache miss

IBM PowerPC RS64-IV (2000)

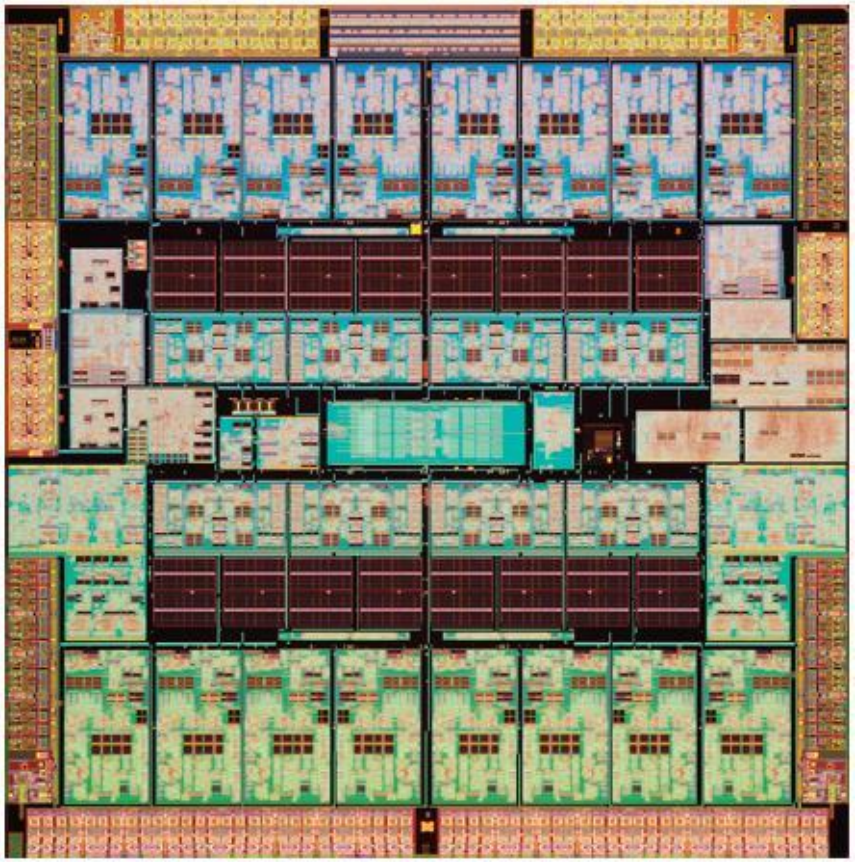
- Commercial coarse-grain multithreading CPU
- Based on PowerPC with quad-issue in-order five-stage pipeline
- Each physical CPU supports two virtual CPUs
- On L2 cache miss, pipeline is flushed and execution switches to second thread
  - short pipeline minimizes flush penalty (4 cycles), small compared to memory access latency
  - flush pipeline to simplify exception handling

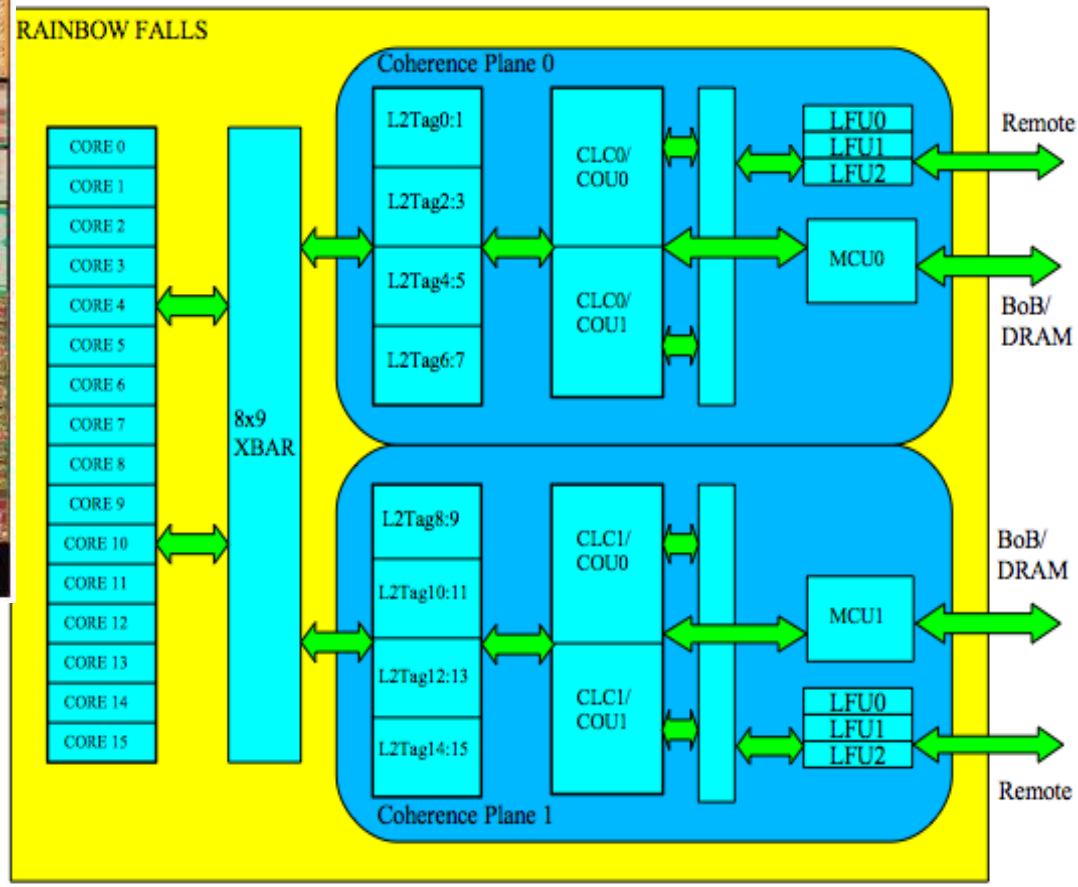
Oracle/Sun Niagara processors

- Target is datacenters running web servers and databases, with many concurrent requests
- Provide multiple simple cores each with multiple hardware threads, reduced energy/operation though much lower single thread performance

- Niagara-1 [2004], 8 cores, 4 threads/core
- Niagara-2 [2007], 8 cores, 8 threads/core
- Niagara-3 [2009], 16 cores, 8 threads/core
- T4 [2011], 8 cores, 8 threads/core
- T5 [2012], 16 cores, 8 threads/core
- M5 [2012], 6 cores, 8 threads/core
- M6 [2013], 12 cores, 8 threads/core

Oracle/Sun Niagara-3, "Rainbow Falls" 2009





Oracle SPARC M6 Processor (2013)

The Next Oracle Processor: SPARC M6

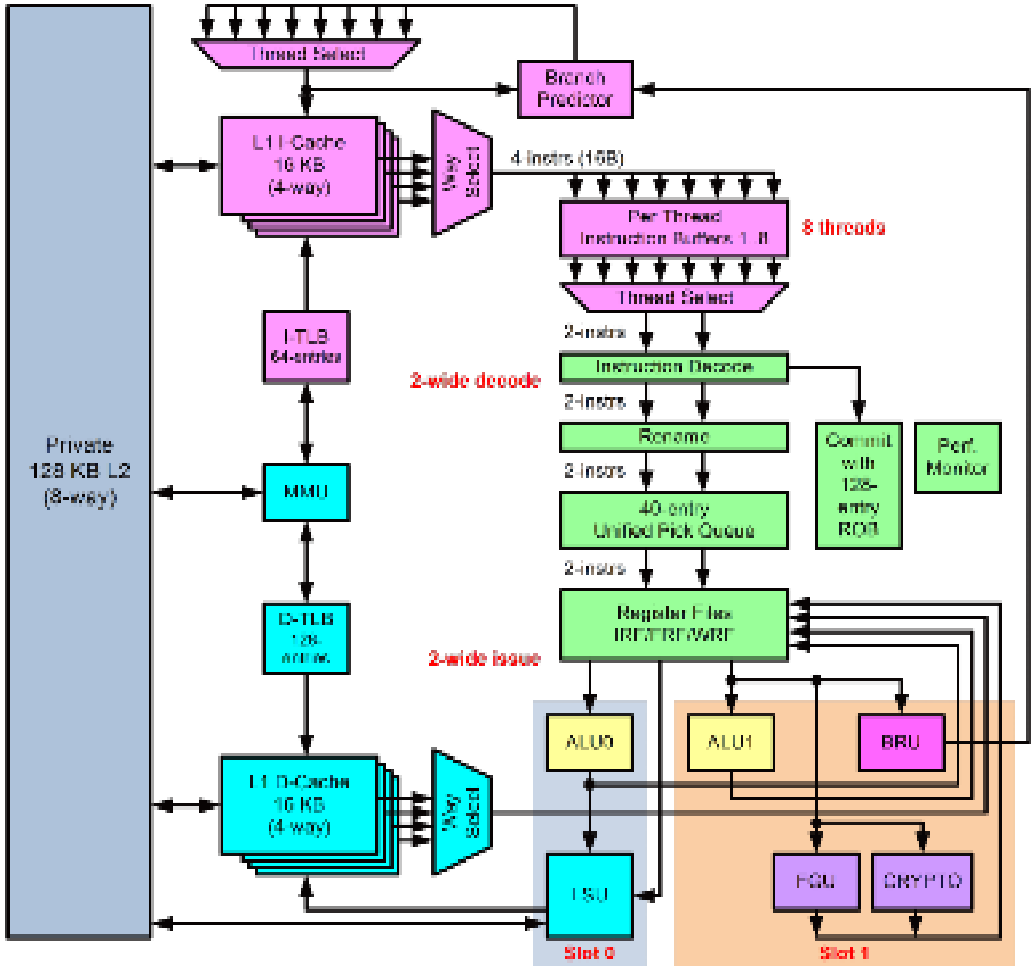
	nm	Cores	Threads	L3$	Memory per Socket	PCIe	Max. Sockets
T4	40	8	64	4MB	0.5TB	2*G2	4
T5	28	16	128	8MB	0.5TB	2*G3	8
M5	28	6	48	48MB	1TB	2*G3	32
M6	28	12	96	48MB	1TB	2*G3	96

ORACLE

Oracle SPARC M6 Core (2013)

SPARC S3 Core

- Dual-issue, out-of-order
- Integrated encryption acceleration instructions
- Enhanced instruction set to accelerate Oracle SW stack
- 1-8 strands, dynamically threaded pipeline

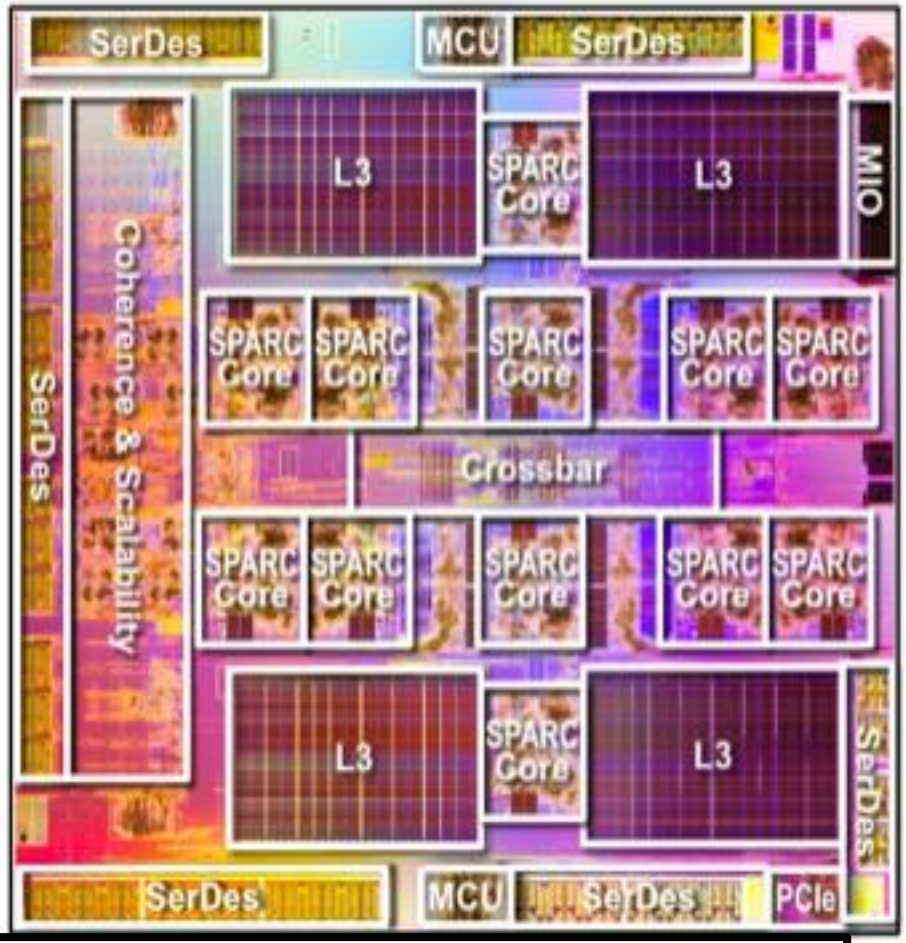


 | |

Oracle SPARC M6 (2013)

SPARC M6: Processor Overview

- 12 SPARC S3 cores, 96 threads
- 48MB shared L3 cache
- 4 DDR3 schedulers, maximum of 1TB of memory per socket
- 2 PCIe 3.0 x8 lanes
- Up to 8 sockets glue-less scaling
- Up to 96 sockets glued scaling
- 4.1 Tbps total link bandwidth
- 4.27 billion transistors

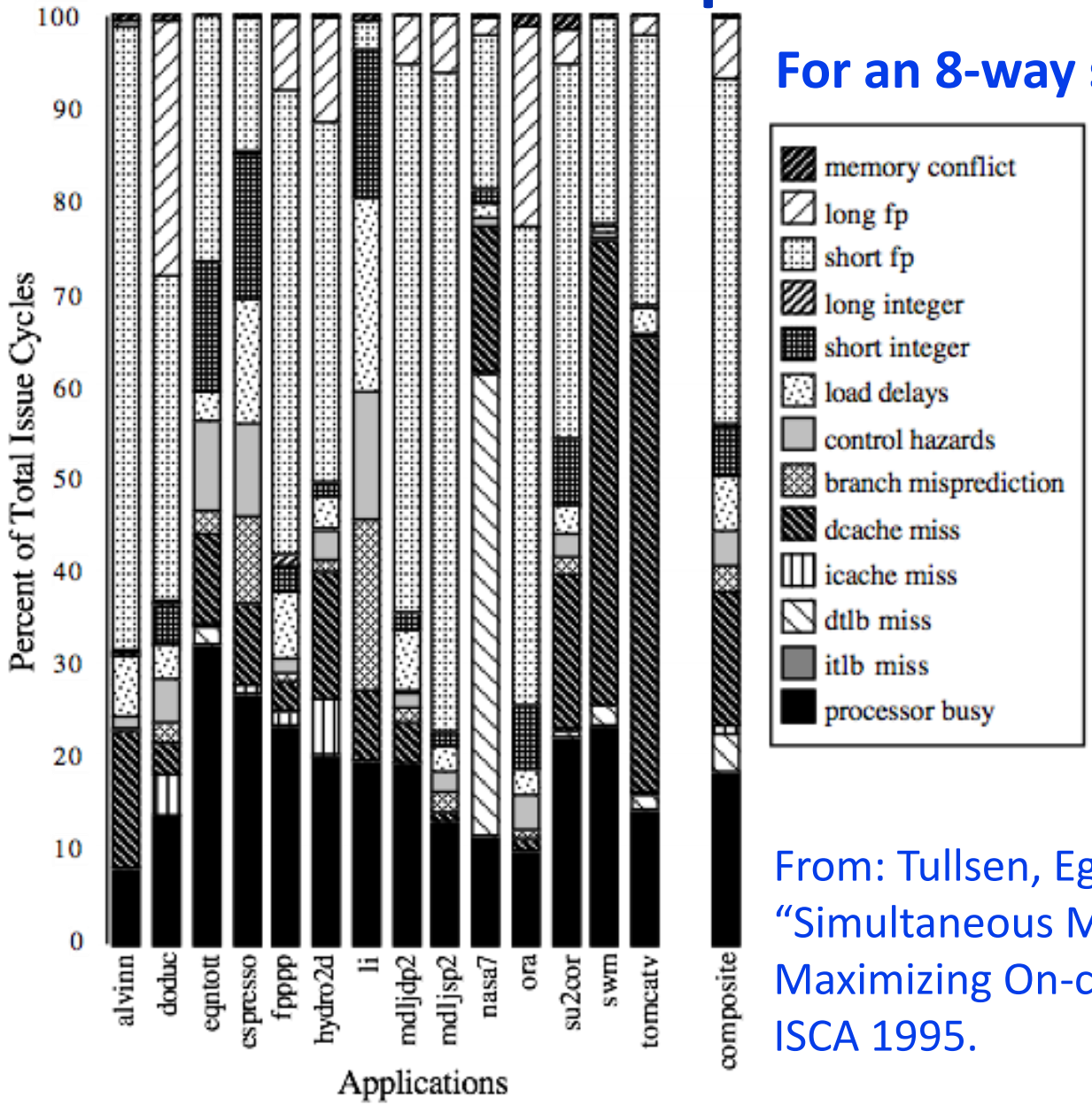


Oracle ended SPARC programs after M8 in 2017

Simultaneous Multithreading (SMT) for OoO Superscalars

- Techniques presented so far have all been “vertical” multithreading where each pipeline stage works on one thread at a time
- SMT uses fine-grain control already present inside an OoO superscalar to allow instructions from multiple threads to enter execution on same clock cycle. Gives better utilization of machine resources.

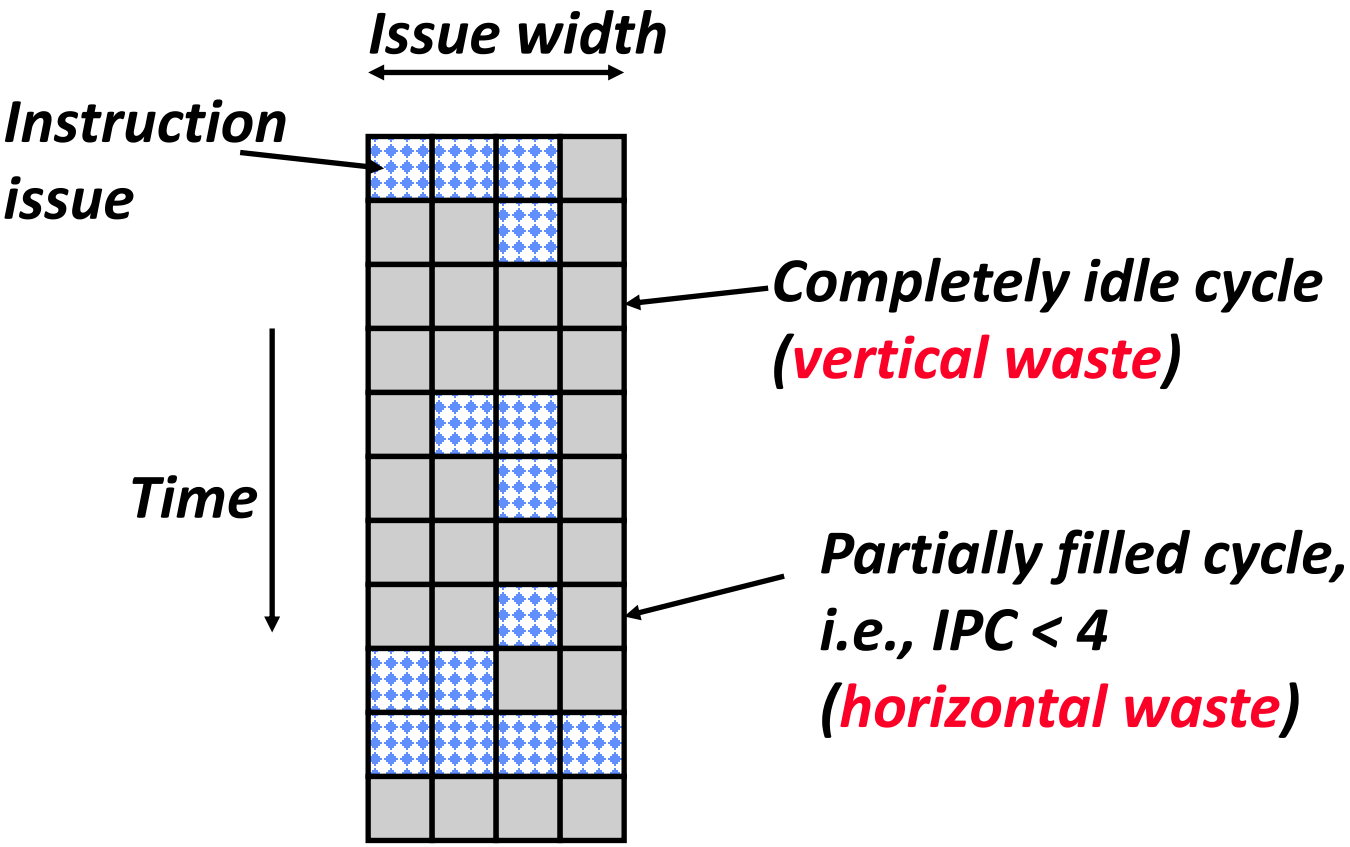
For most apps, most execution units lie idle in an OoO superscalar



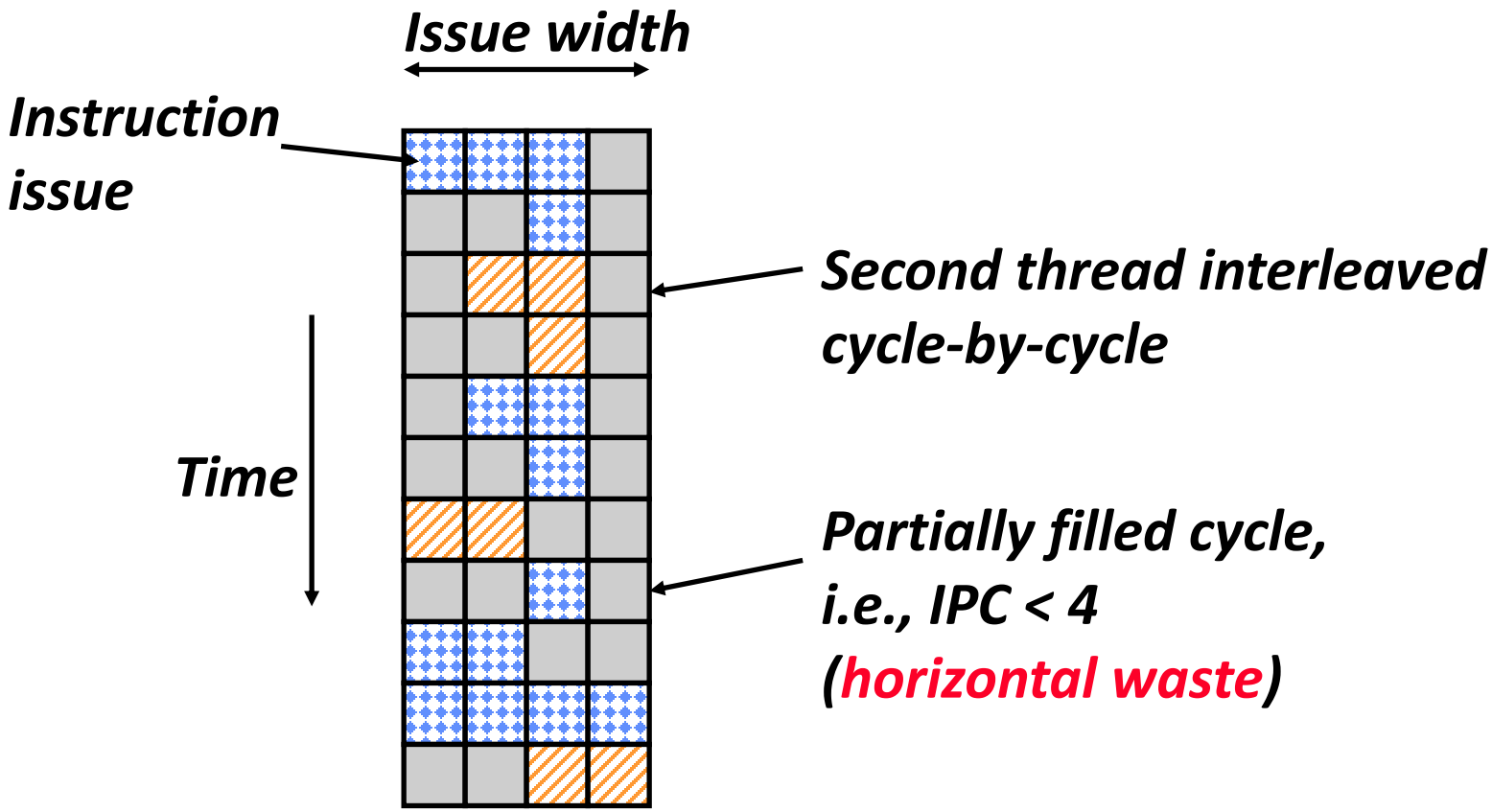
For an 8-way superscalar.

From: Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism", ISCA 1995.

Superscalar Machine Efficiency

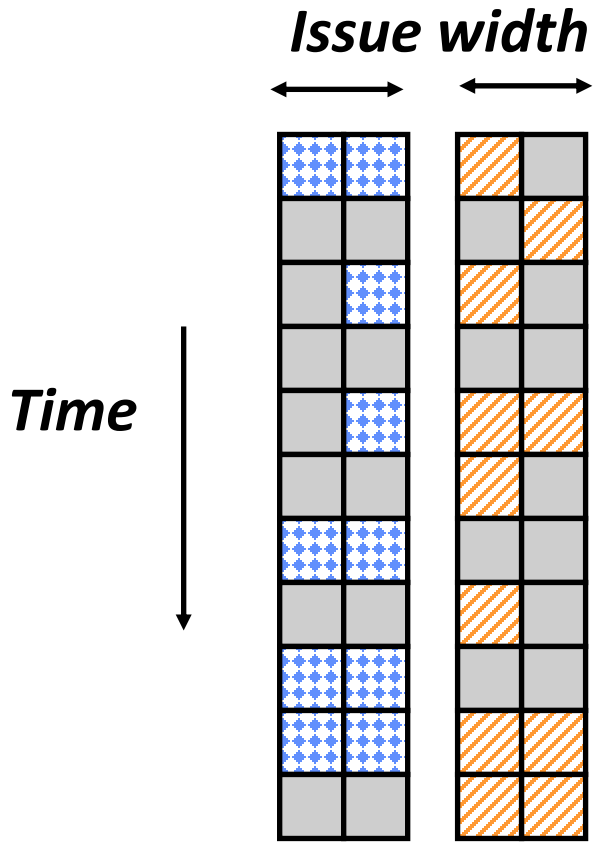


Vertical Multithreading



- Cycle-by-cycle interleaving removes vertical waste, but leaves some horizontal waste

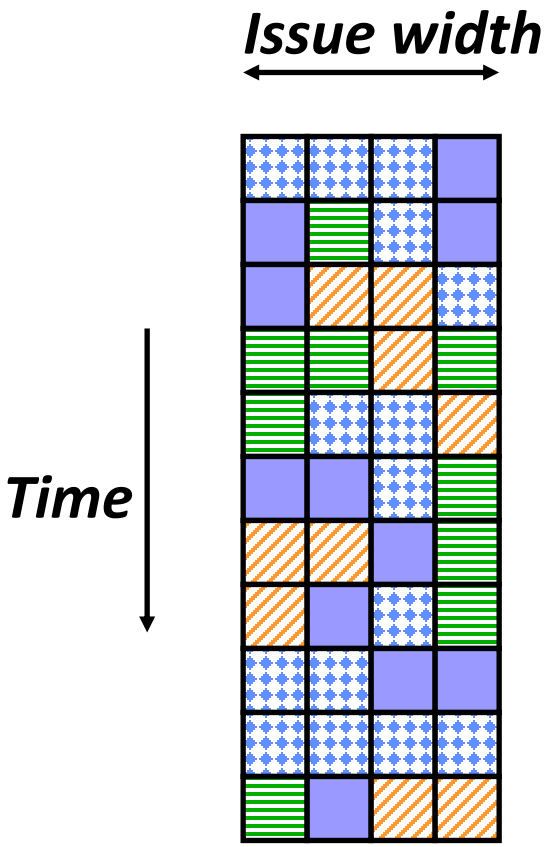
Chip Multiprocessing (CMP)



- What is the effect of splitting into multiple processors?
  - reduces horizontal waste,
  - leaves some vertical waste, and
  - puts upper limit on peak throughput of each thread.

Ideal Superscalar Multithreading

[Tullsen, Eggers, Levy, UW, 1995]



- Interleave multiple threads to multiple issue slots with no restrictions

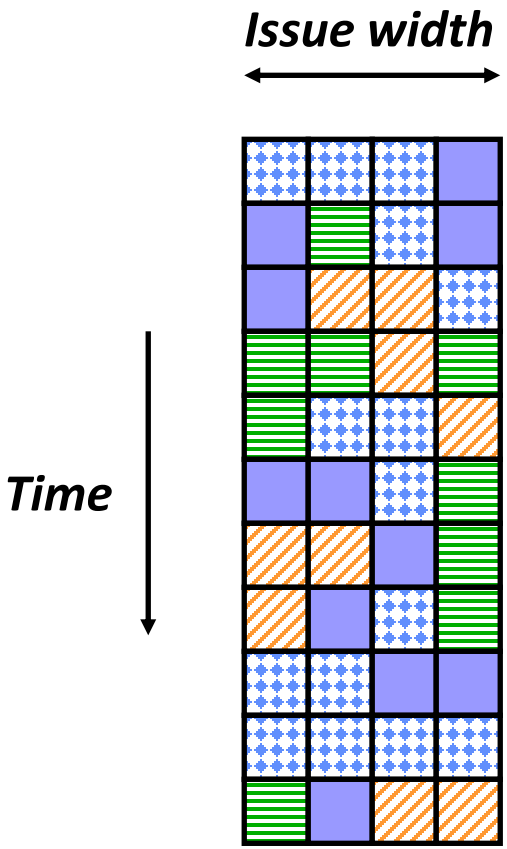
O-o-O Simultaneous Multithreading

[Tullsen, Eggers, Emer, Levy, Stamm, Lo, DEC/UW, 1996]

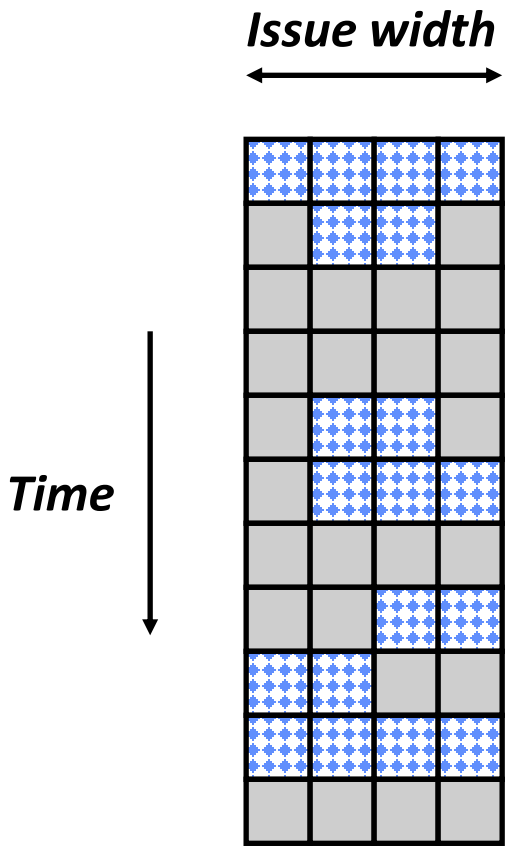
- Add multiple contexts and fetch engines and allow instructions fetched from different threads to issue simultaneously
- Utilize wide out-of-order superscalar processor issue queue to find instructions to issue from multiple threads
- OOO instruction window already has most of the circuitry required to schedule from multiple threads
- Any single thread can utilize whole machine

SMT adaptation to parallelism type

For regions with high thread-level parallelism (TLP) entire machine width is shared by all threads



For regions with low thread-level parallelism (TLP) entire machine width is available for instruction-level parallelism (ILP)

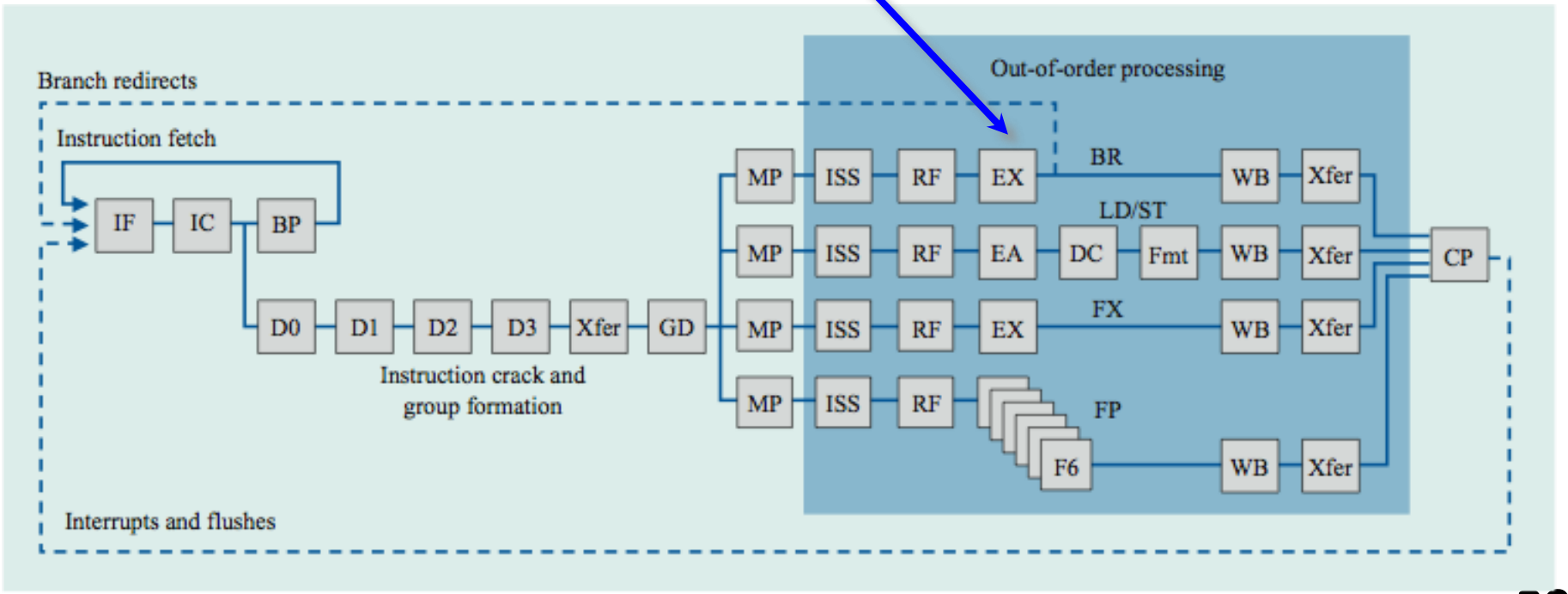


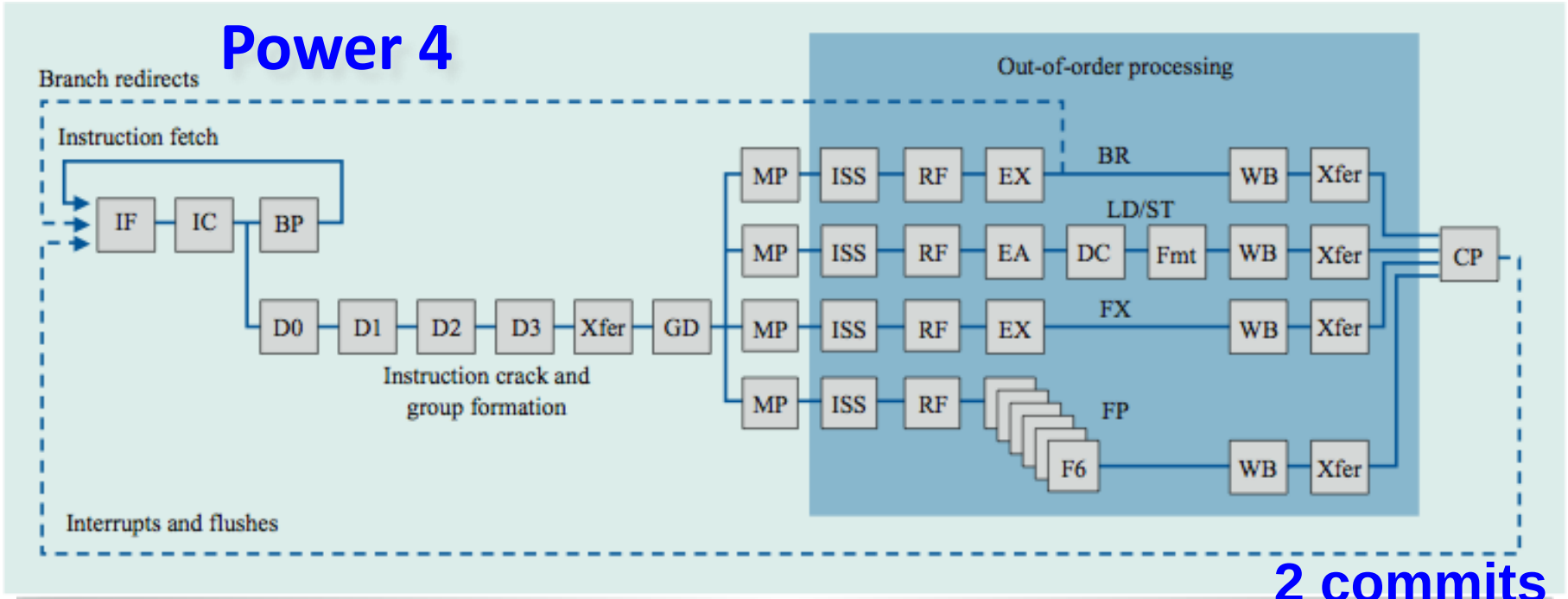
Pentium-4 Hyperthreading (2002)

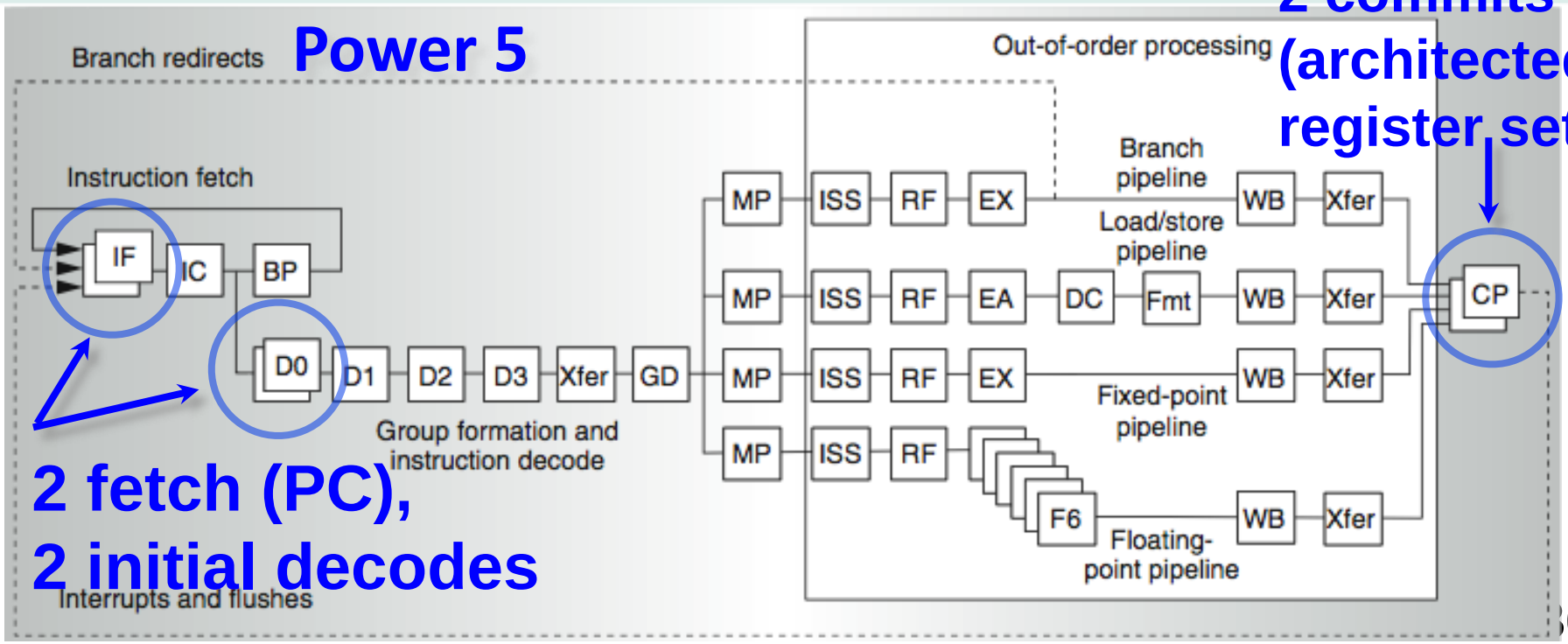
- First commercial SMT design (2-way SMT)
- Logical processors share nearly all resources of the physical processor
  - Caches, execution units, branch predictors
- Die area overhead of hyperthreading ~ 5%
- When one logical processor is stalled, the other can make progress
  - No logical processor can use all entries in queues when two threads are active
- Processor running only one active software thread runs at approximately same speed with or without hyperthreading
- Hyperthreading dropped on OoO P6-based followons to Pentium-4 (Pentium-M, Core Duo, Core 2 Duo), until revived with Nehalem generation machines in 2008.
- First Intel Atom (in-order x86 core) has two-way vertical multithreading
  - Hyperthreading == (SMT for Intel OoO & Vertical for Intel InO)

IBM Power 4

Single-threaded predecessor to Power 5.
8 execution units in out-of-order engine,
each may issue an instruction each cycle.



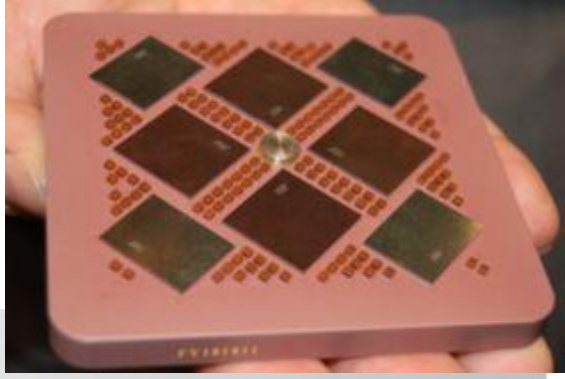


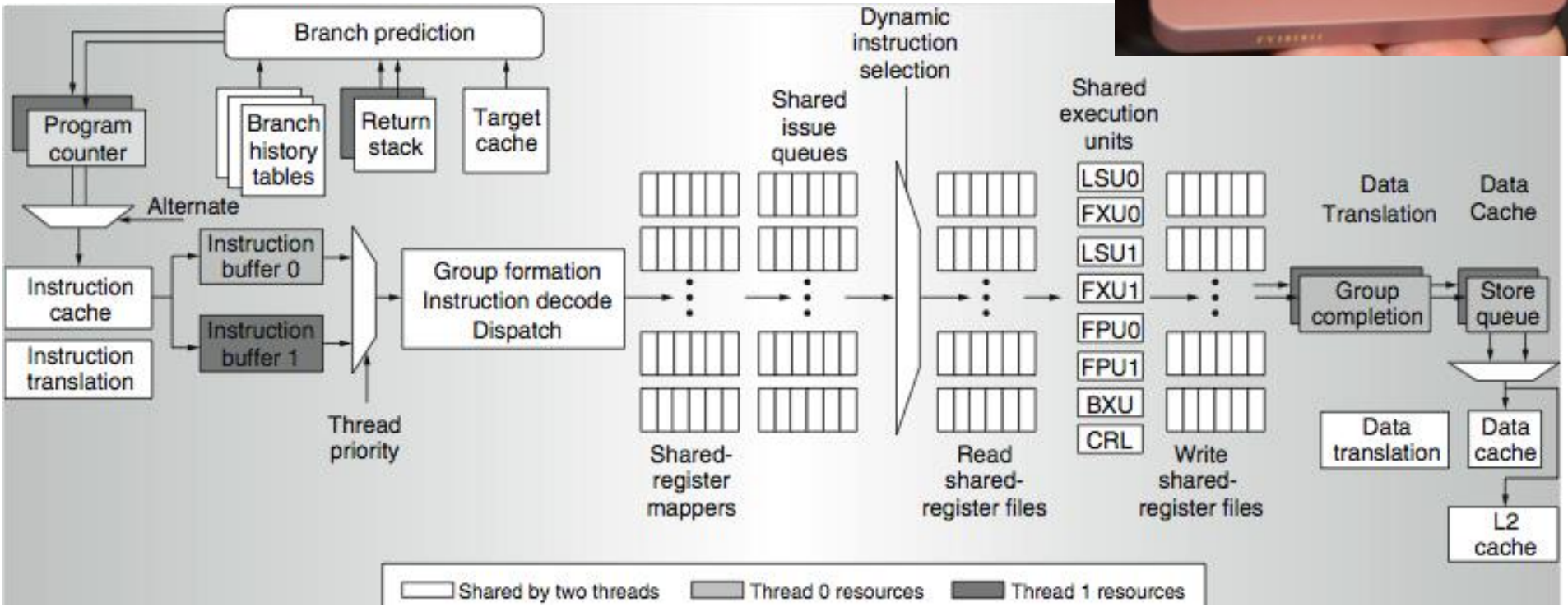


2 commits (architected register sets)

2 fetch (PC),
2 initial decodes

Power 5 data flow ...





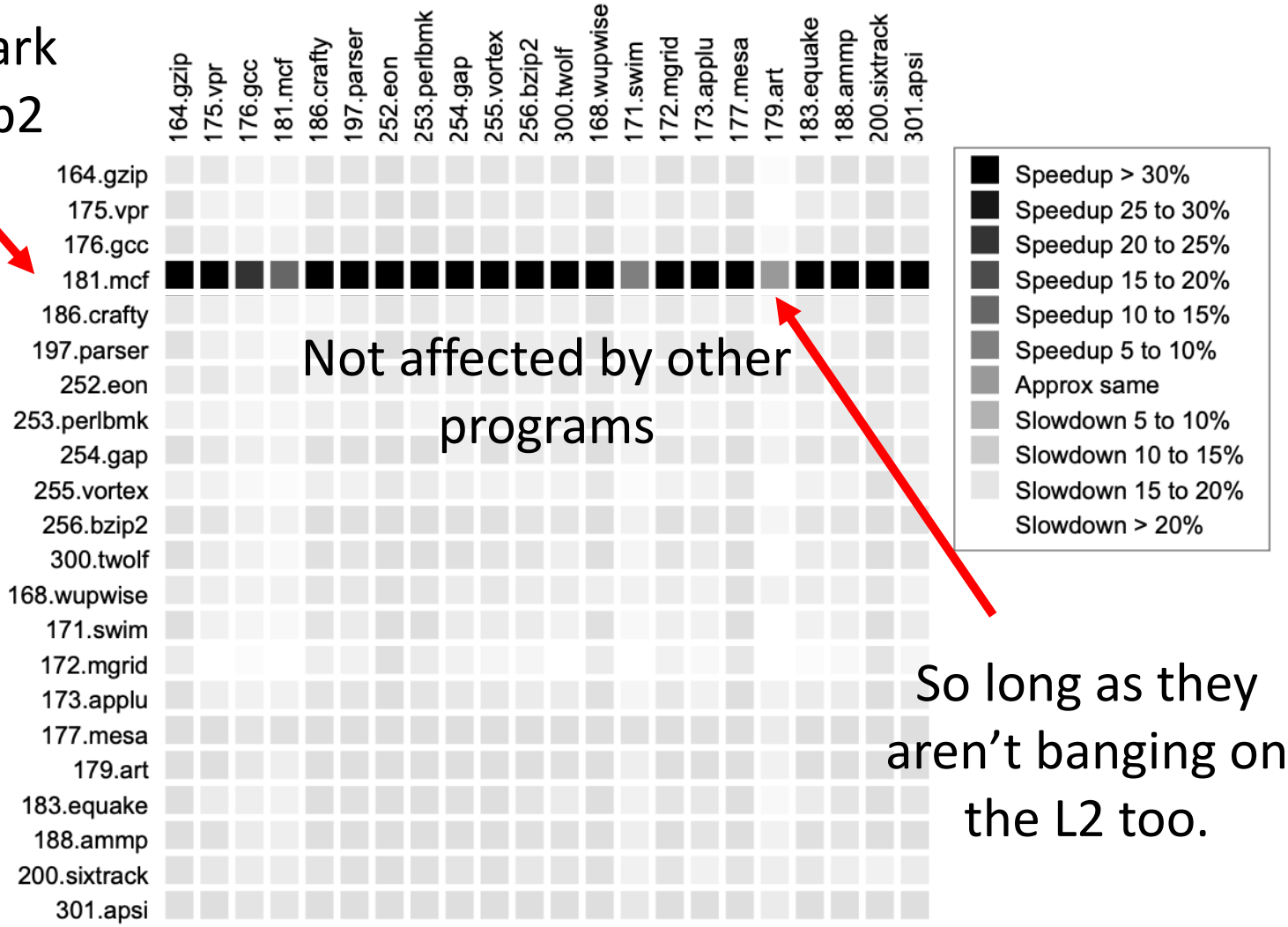
Why only 2 threads? With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck

Initial Performance of SMT

- Pentium-4 Extreme SMT yields 1.01 speedup for SPECint_rate benchmark and 1.07 for SPECfp_rate
  - Pentium-4 is dual-threaded SMT
  - SPECRate requires that each SPEC benchmark be run against a vendor-selected number of copies of the same benchmark
- Running on Pentium-4 each of 26 SPEC benchmarks paired with every other (26^2 runs) speed-ups from 0.90 to 1.58; average was 1.20
- Power 5, 8-processor server 1.23 faster for SPECint_rate with SMT, 1.16 faster for SPECfp_rate
- Power 5 running 2 copies of each app speedup between 0.89 and 1.41
  - Most gained some
  - Fl.Pt. apps had most cache conflicts and least gains

SMT Performance: Application Interaction

Your favorite benchmark from Lab2



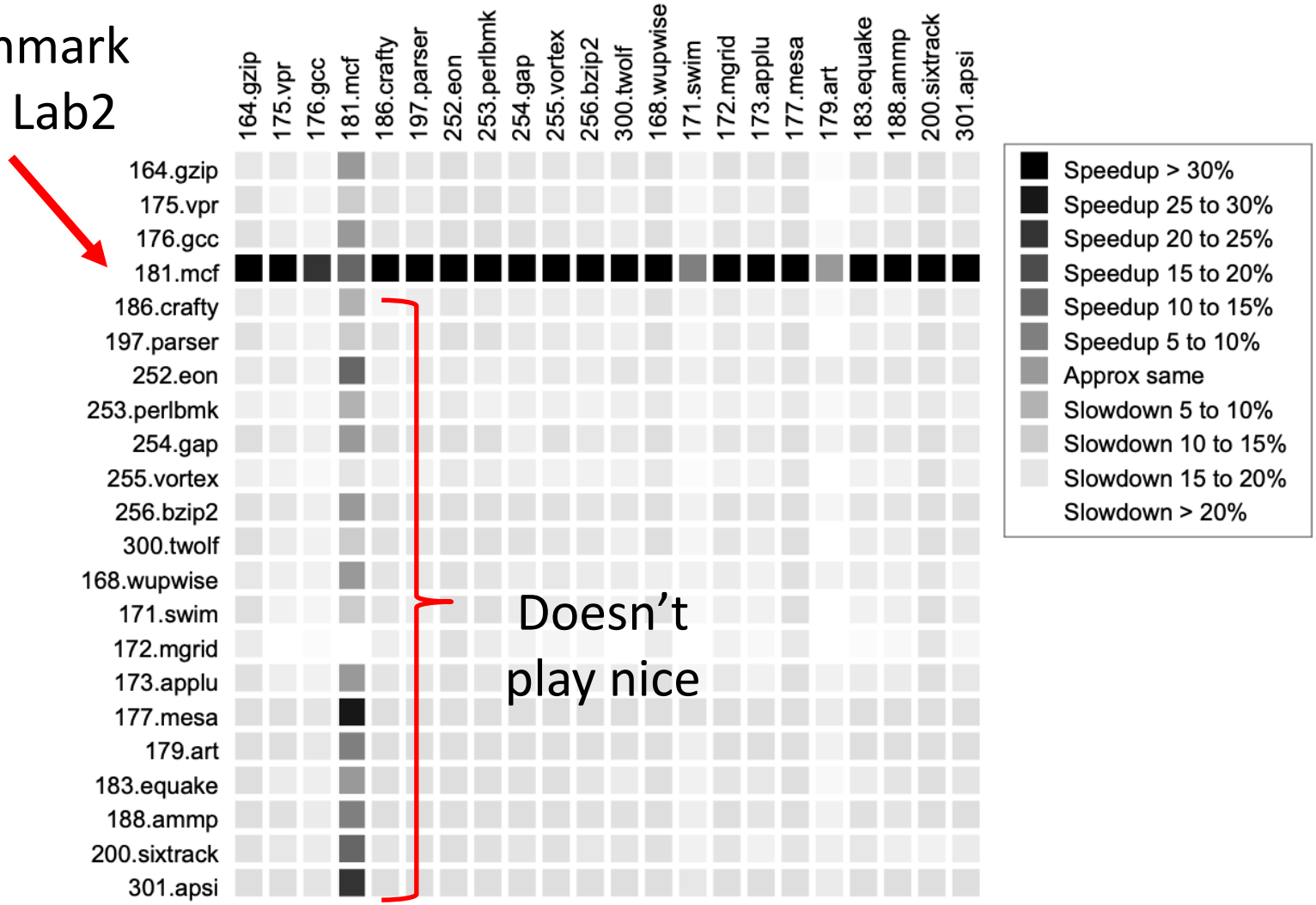
Not affected by other programs

So long as they aren't banging on the L2 too.

Bulpin et al, "*Multiprogramming Performance of Pentium 4 with Hyper-Threading*"

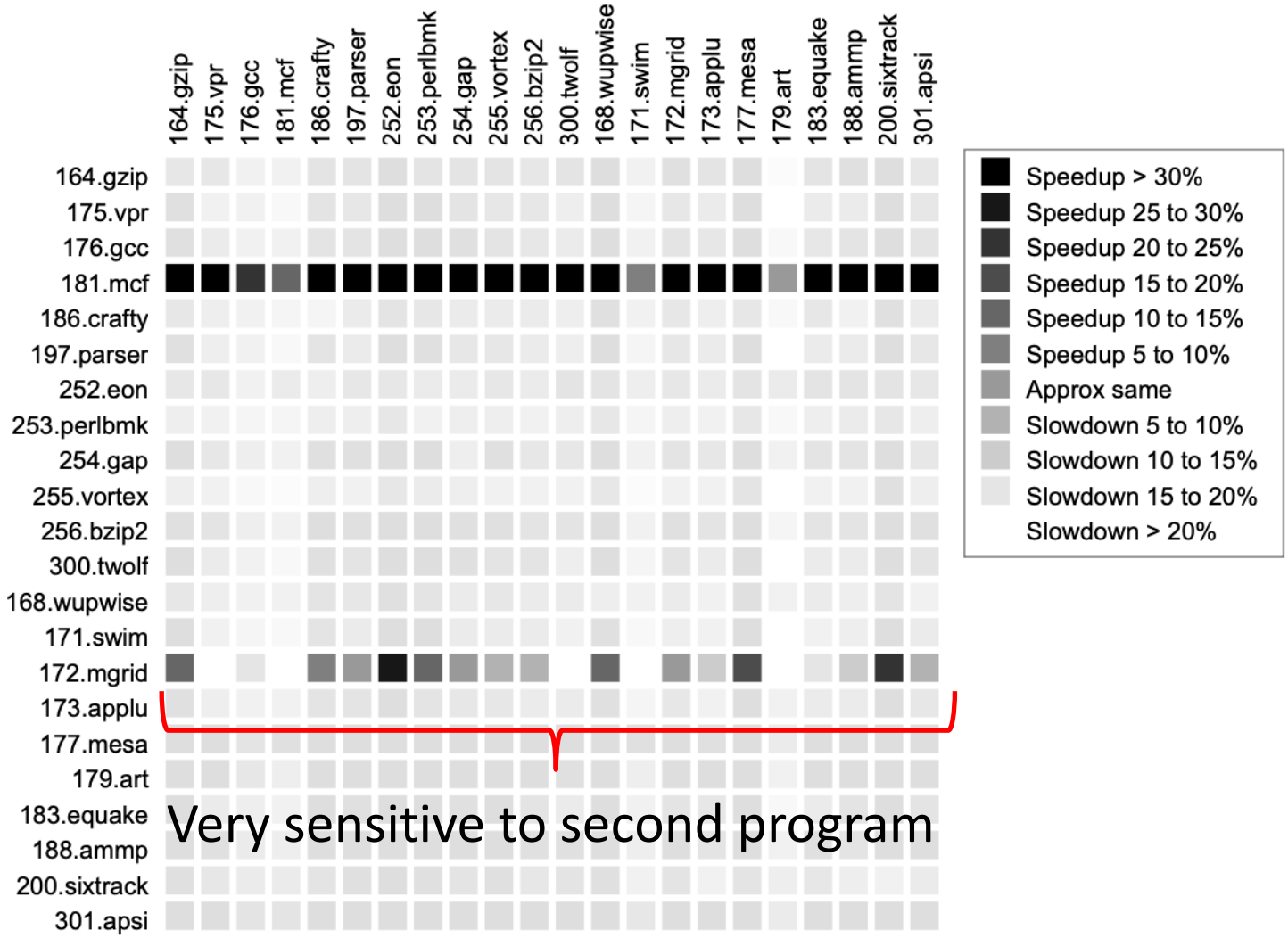
SMT Performance: Application Interaction

Your favorite benchmark from Lab2



Bulpin et al, "*Multiprogramming Performance of Pentium 4 with Hyper-Threading*"

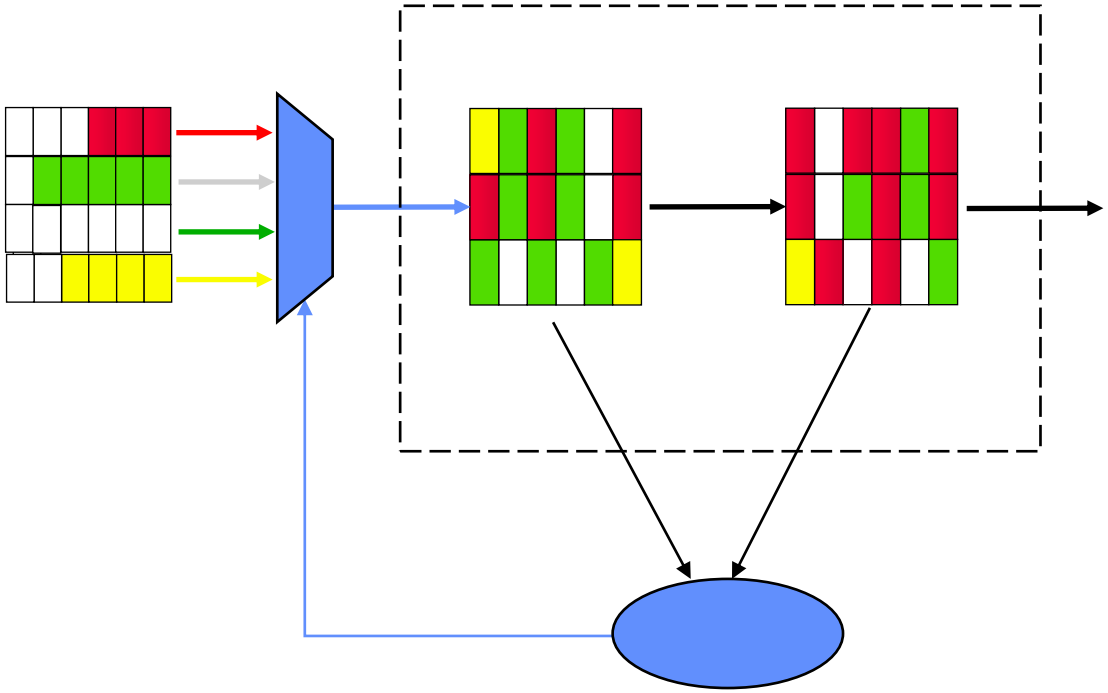
SMT Performance: Application Interaction



Bulpin et al, "*Multiprogramming Performance of Pentium 4 with Hyper-Threading*"

Icount Choosing Policy

Fetch from thread with the least instructions in flight.



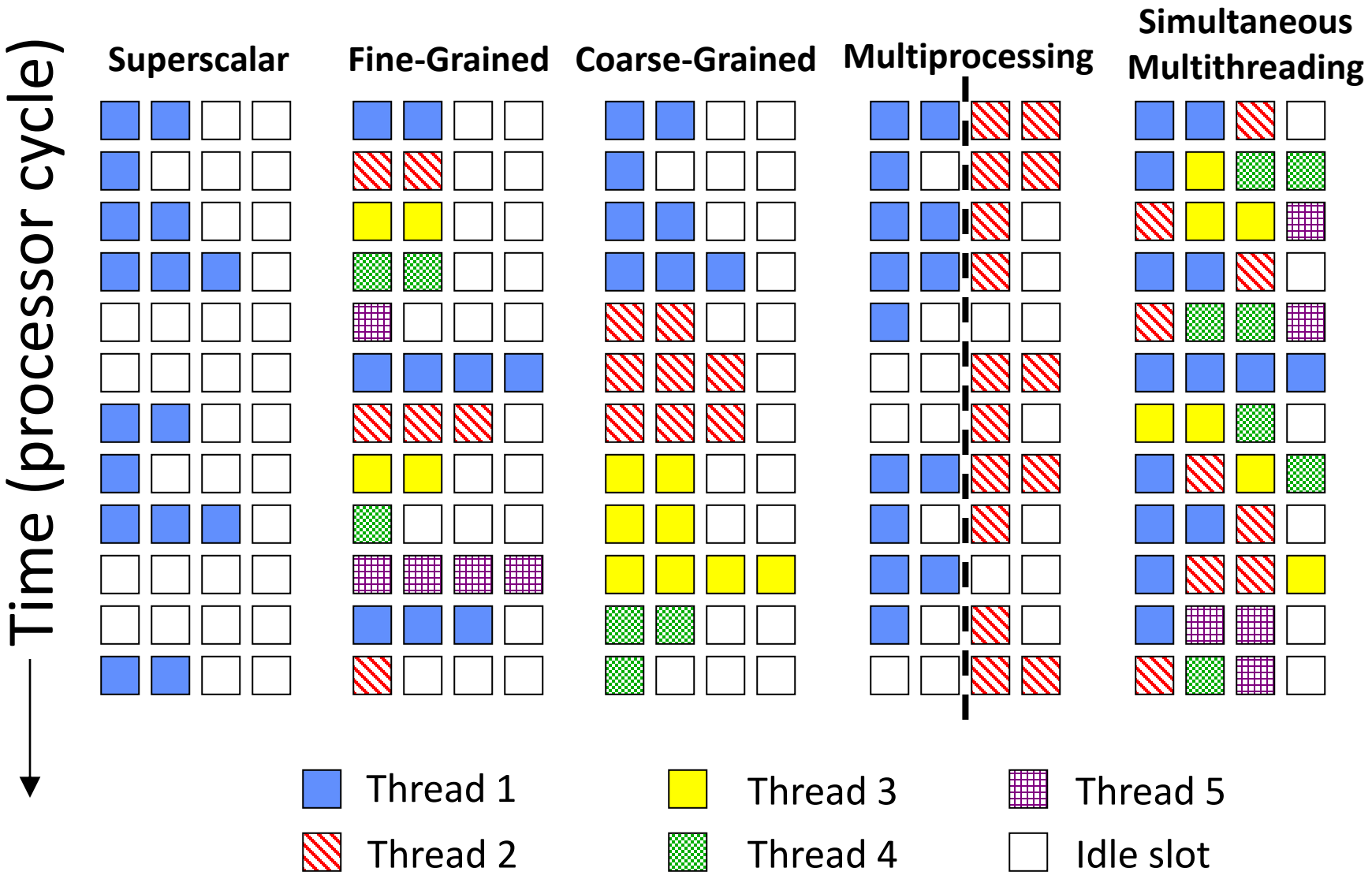
Why does this enhance throughput?

SMT & Security

- Most hardware attacks rely on shared hardware resources to establish a side-channel
  - Eg. Shared outer caches, DRAM row buffers
- SMT gives attackers high-BW access to previously private hardware resources that are shared by co-resident threads:
- TLBs: TLBleed (June, ‘18)
- L1 caches: CacheBleed (2016)
- Functional unit ports: PortSmash (Nov, ’18)

OpenBSD 6.4 → Disabled HT in BIOS, AMD SMT to follow

Summary: Multithreaded Categories



CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 09-2– Vectors

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

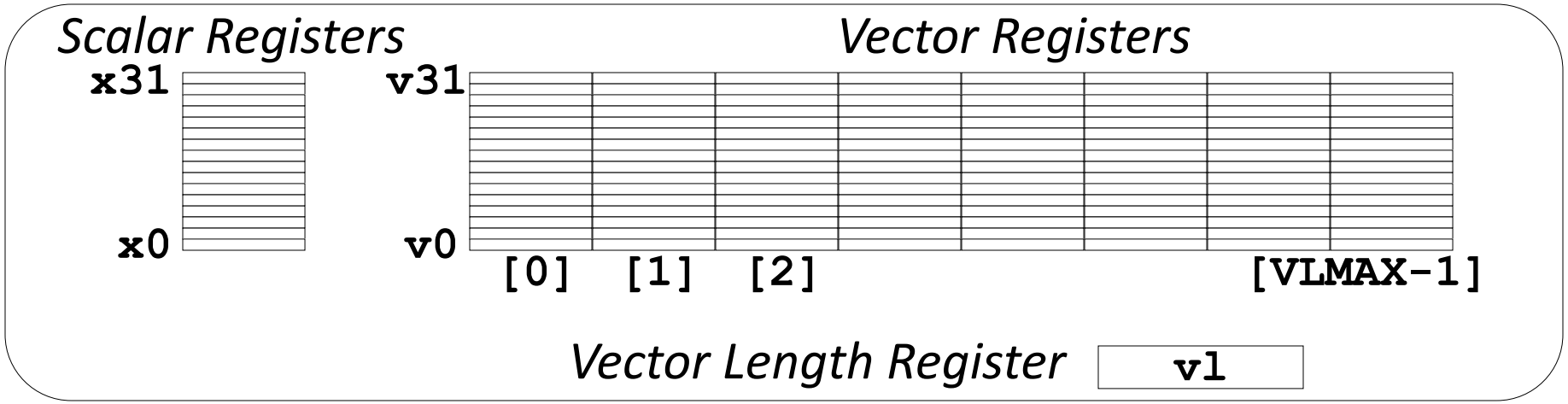
Supercomputer Applications

- Typical application areas
  - Military research (nuclear weapons, cryptography)
  - Scientific research
  - Weather forecasting
  - Oil exploration
  - Industrial design (car crash simulation)
  - Bioinformatics
  - Cryptography
- All involve huge computations on large data set
- Supercomputers: CDC6600, CDC7600, Cray-1, …
- In 70s-80s, Supercomputer ≡ Vector Machine

Vector Supercomputers

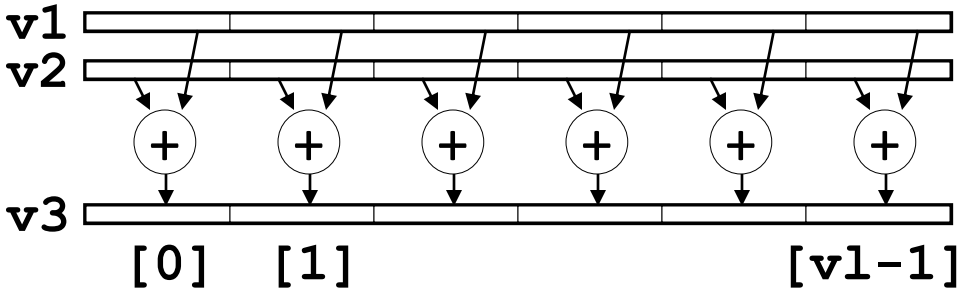
- Epitomized by Cray-1, 1976:
- Scalar Unit
  - Load/Store Architecture
- Vector Extension
  - Vector Registers
  - Vector Instructions
- Implementation
  - Hardwired Control
  - Highly Pipelined Functional Units
  - Interleaved Memory System
  - No Data Caches
  - No Virtual Memory

Vector Programming Model



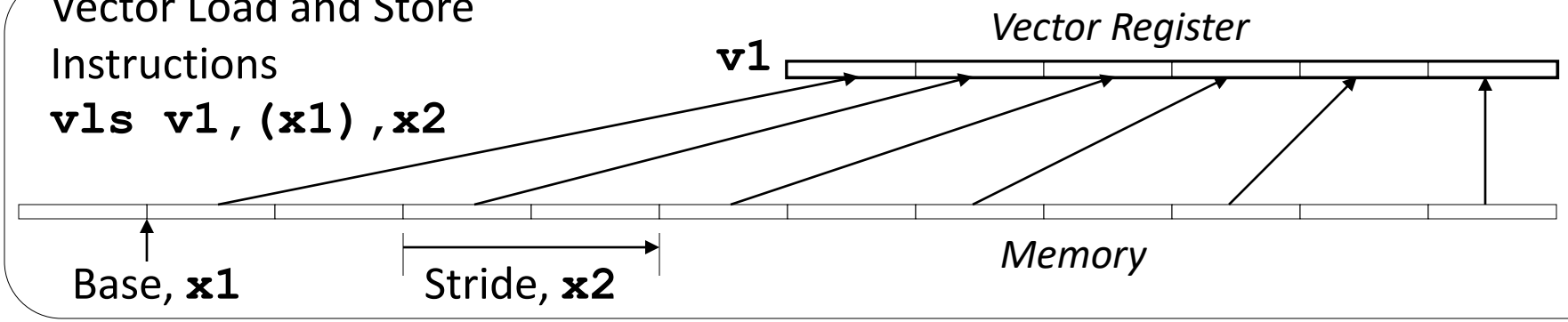
Vector Arithmetic Instructions

`vadd v3,v1,v2`



Vector Load and Store Instructions

`vls v1,(x1),x2`



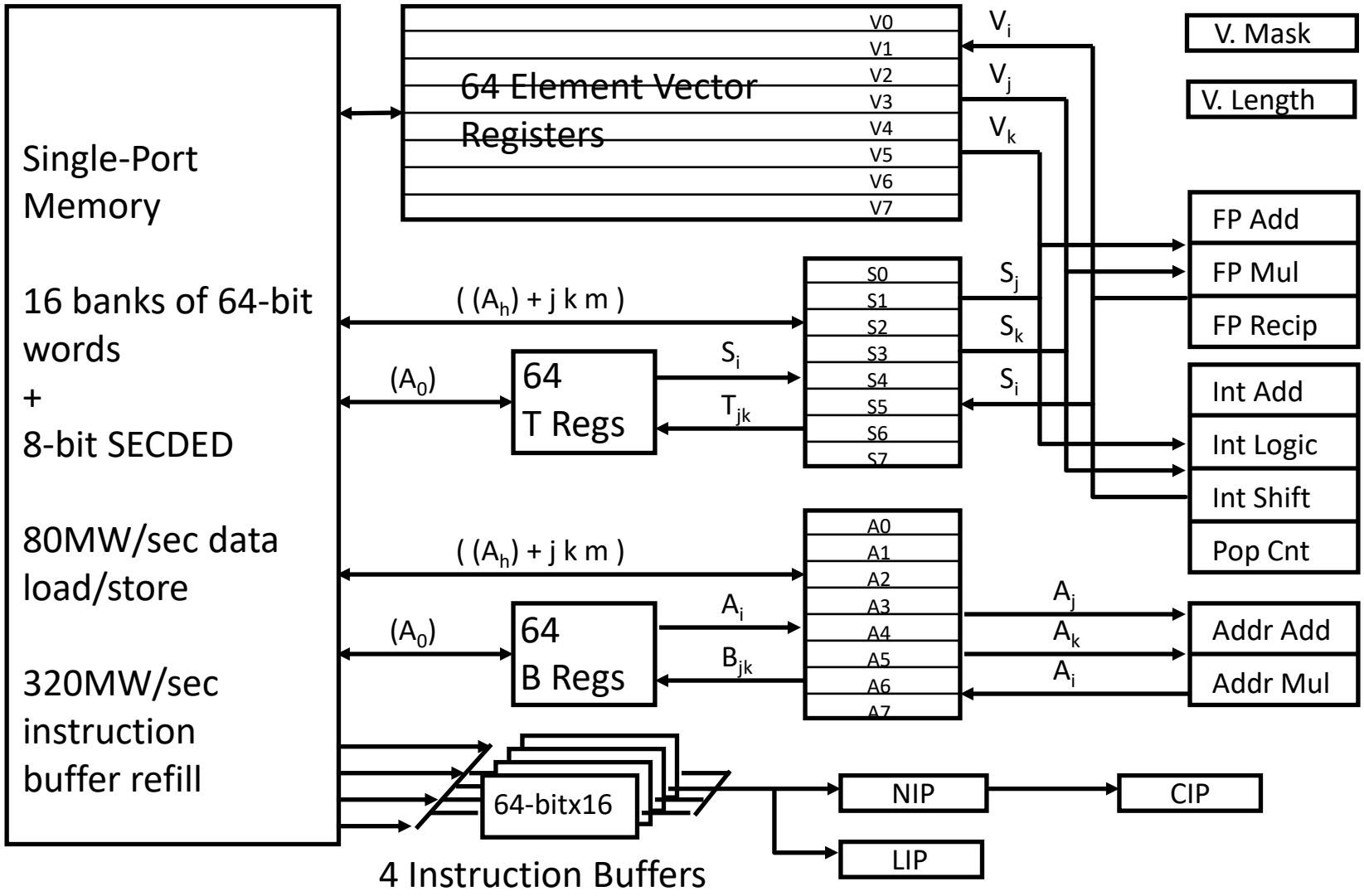
Vector Code Example

```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  li x4, 64
loop:
  fld f1, 0(x1)
  fld f2, 0(x2)
  fadd.d f3,f1,f2
  fsd f3, 0(x3)
  addi x1, x1, 8
  addi x2, x2, 8
  addi x3, x3, 8
  subi x4, x4, 1
  bnez x4, loop
```

```
# Vector Code
  li x4, 64
  vsetvl x4
  vld v1, (x1)
  vld v2, (x2)
  vadd v3,v1,v2
  vst v3, (x3)
```

Cray-1 (1976)



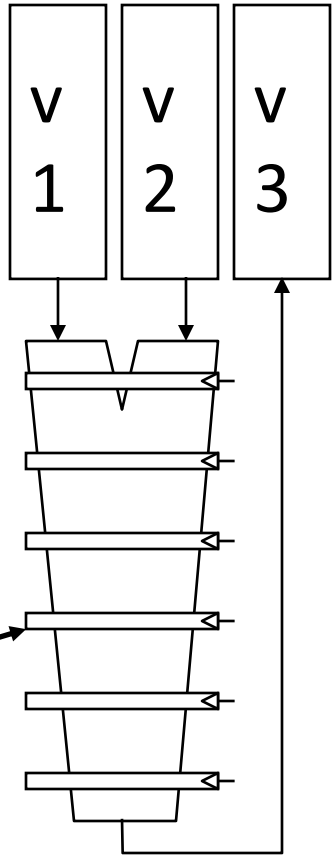
memory bank cycle 50 ns *processor cycle* 12.5 ns (80MHz)

Vector Instruction Set Advantages

- Compact
  - one short instruction encodes N operations
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - access memory in a known pattern (strided load/store)
- Scalable
  - can run same code on more parallel pipelines (lanes)

Vector Arithmetic Execution

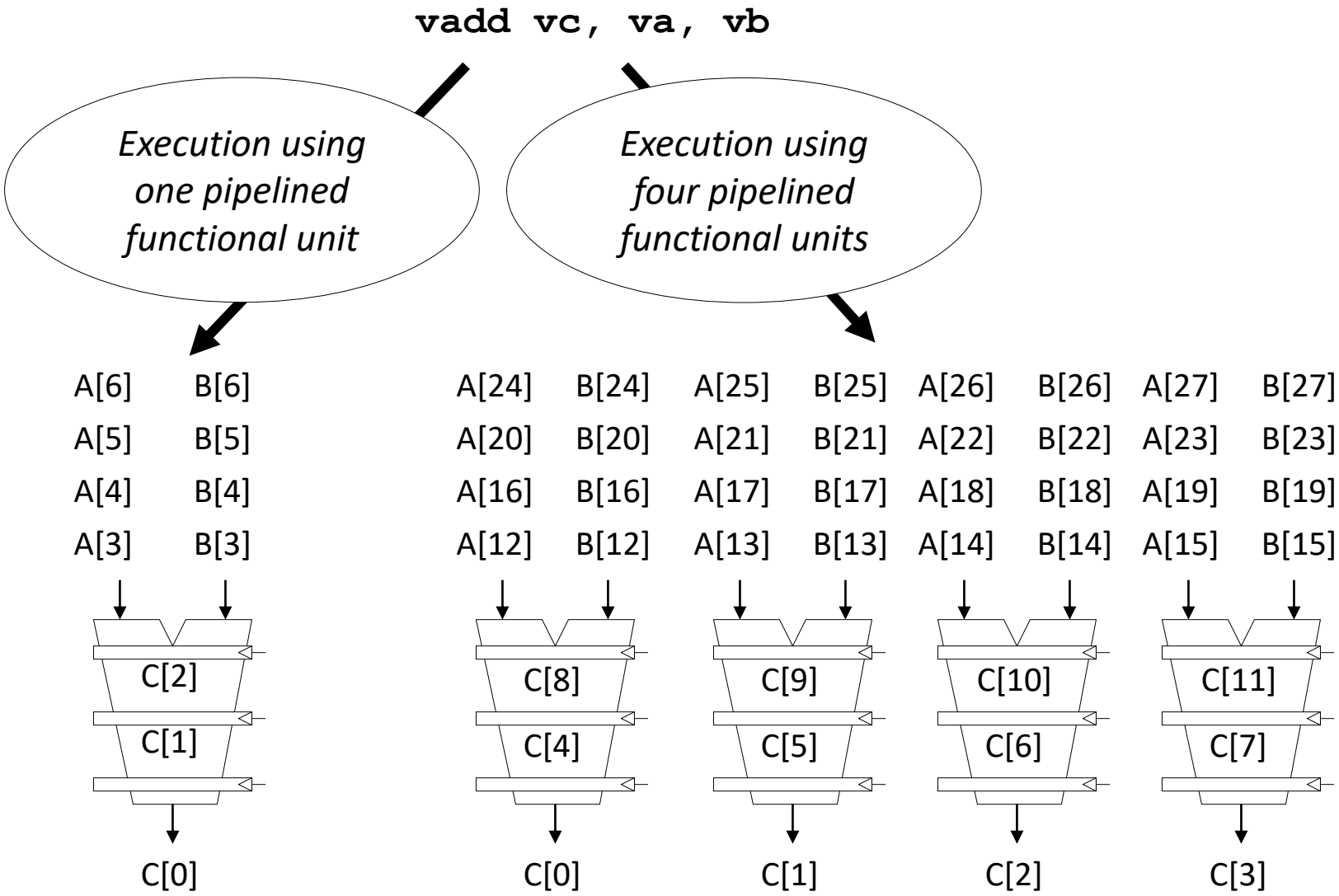
- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)



Six-stage multiply pipeline

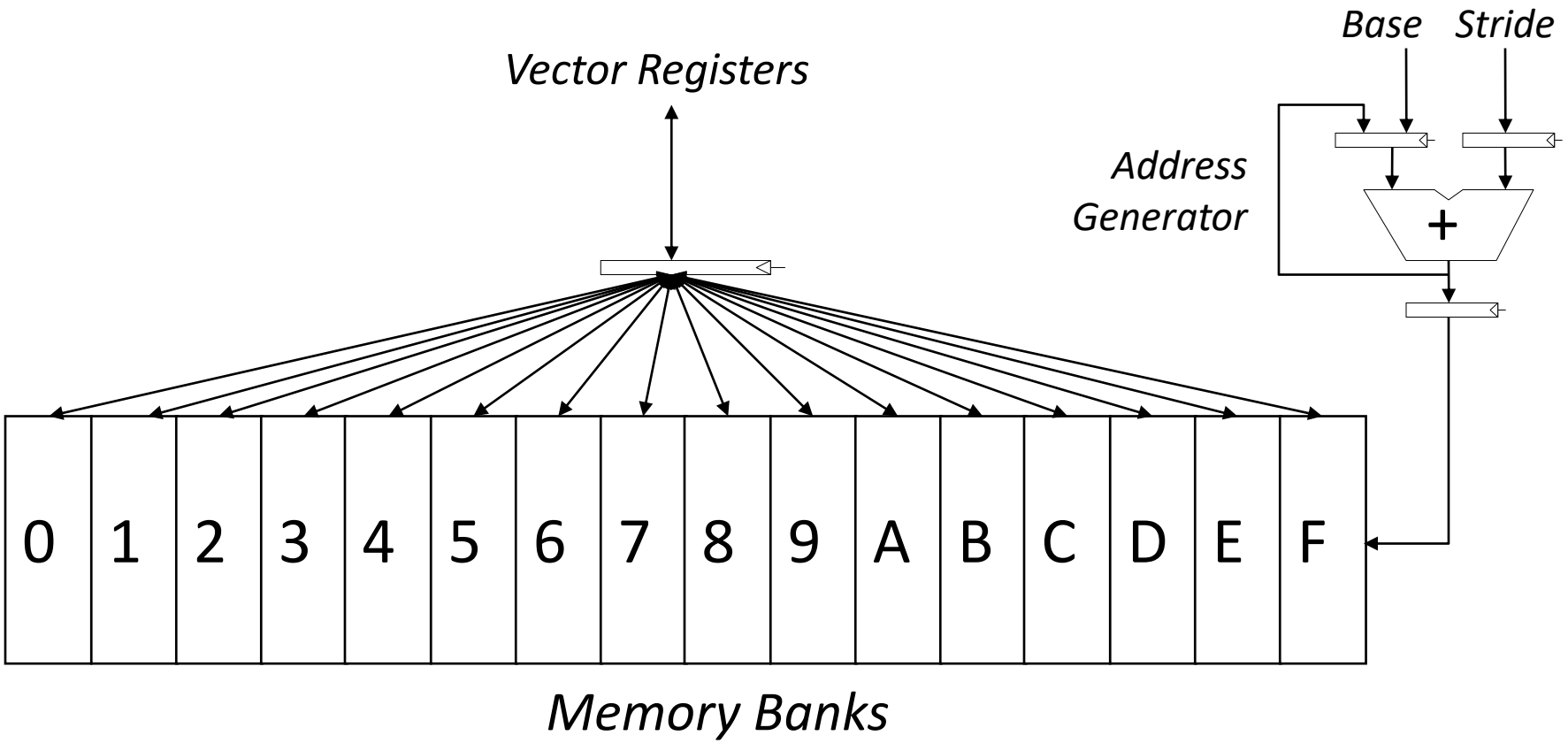
v3 <- v1 * v2

Vector Instruction Execution

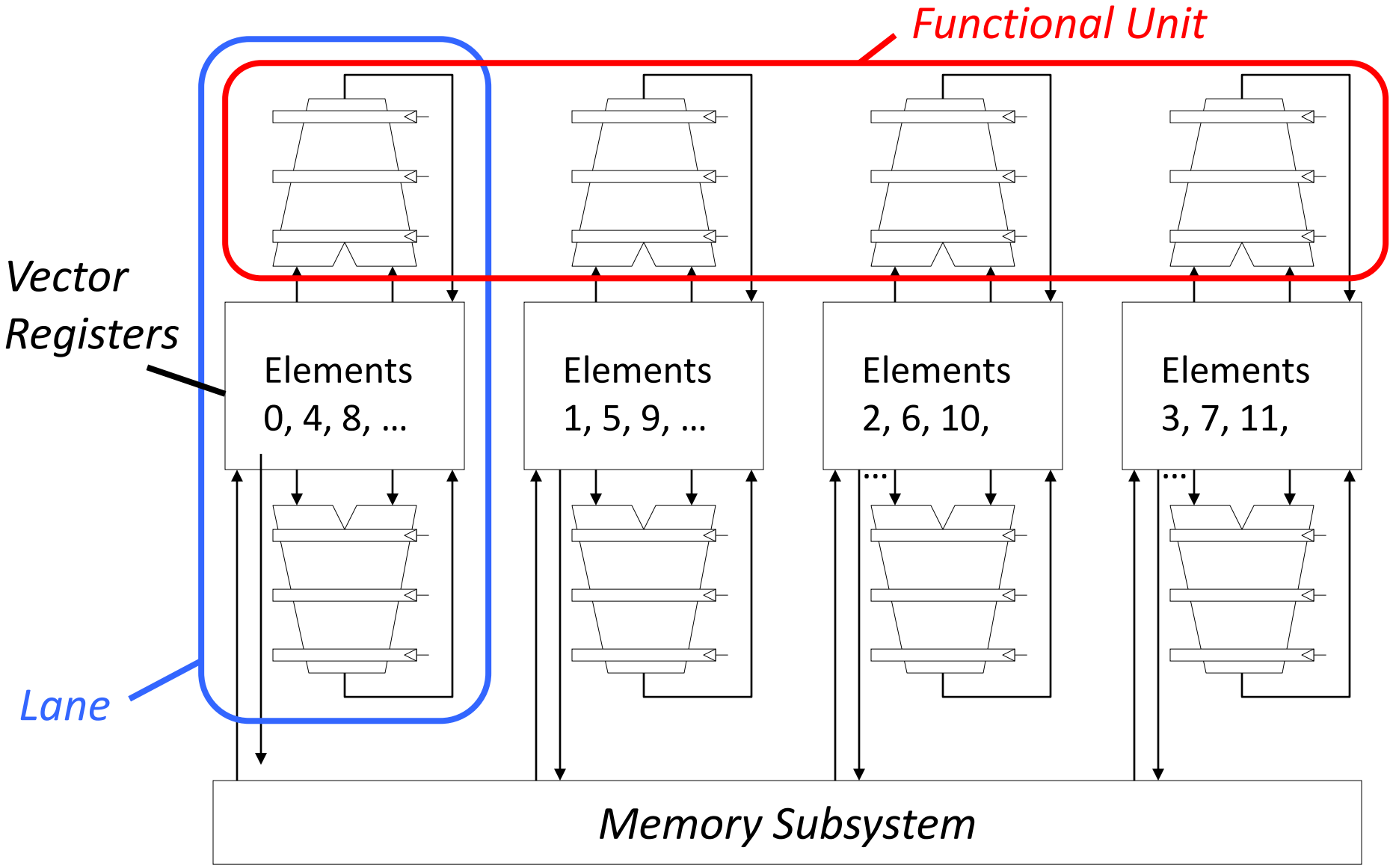


Interleaved Vector Memory System

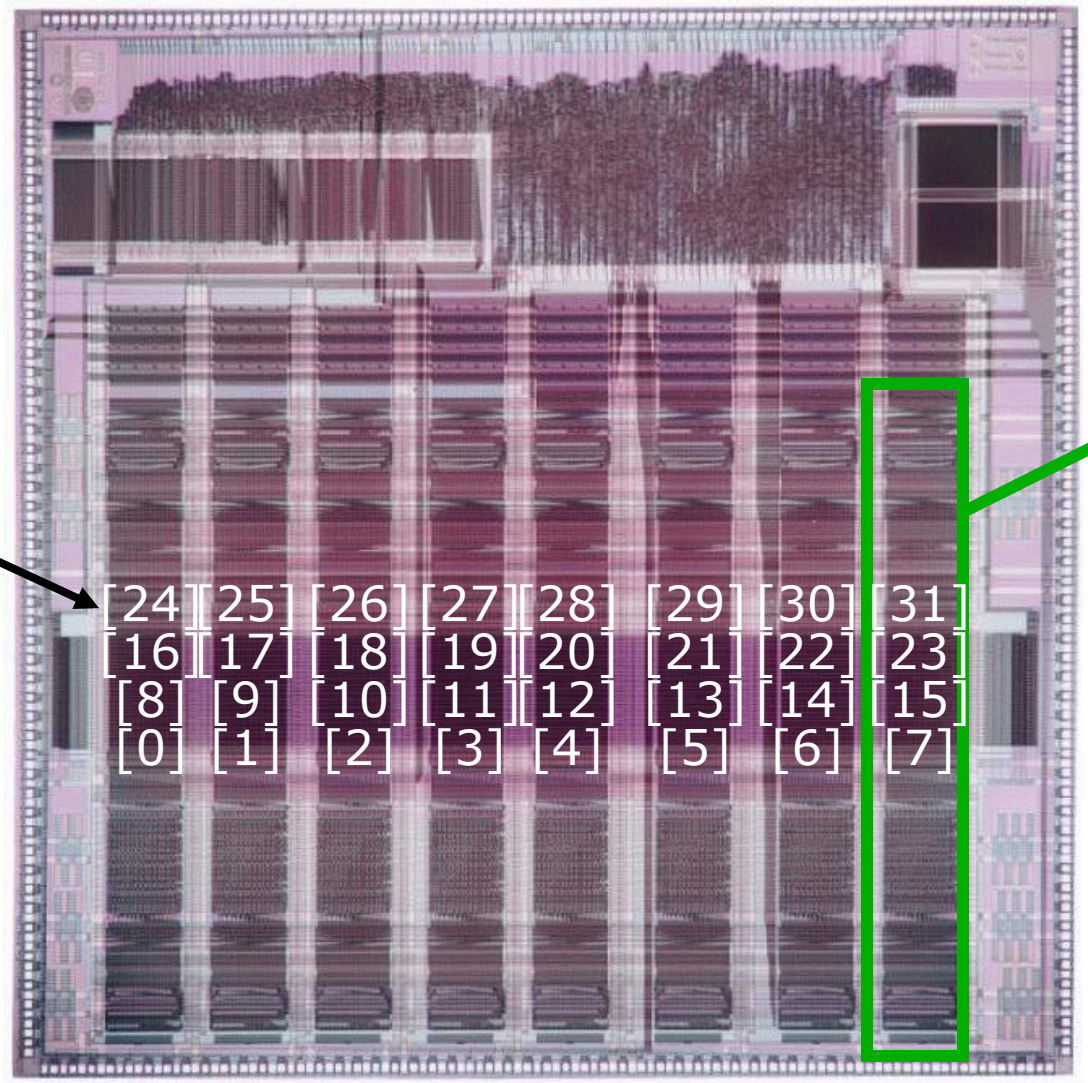
- Bank busy time: Time before bank ready to accept next request
- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency



Vector Unit Structure

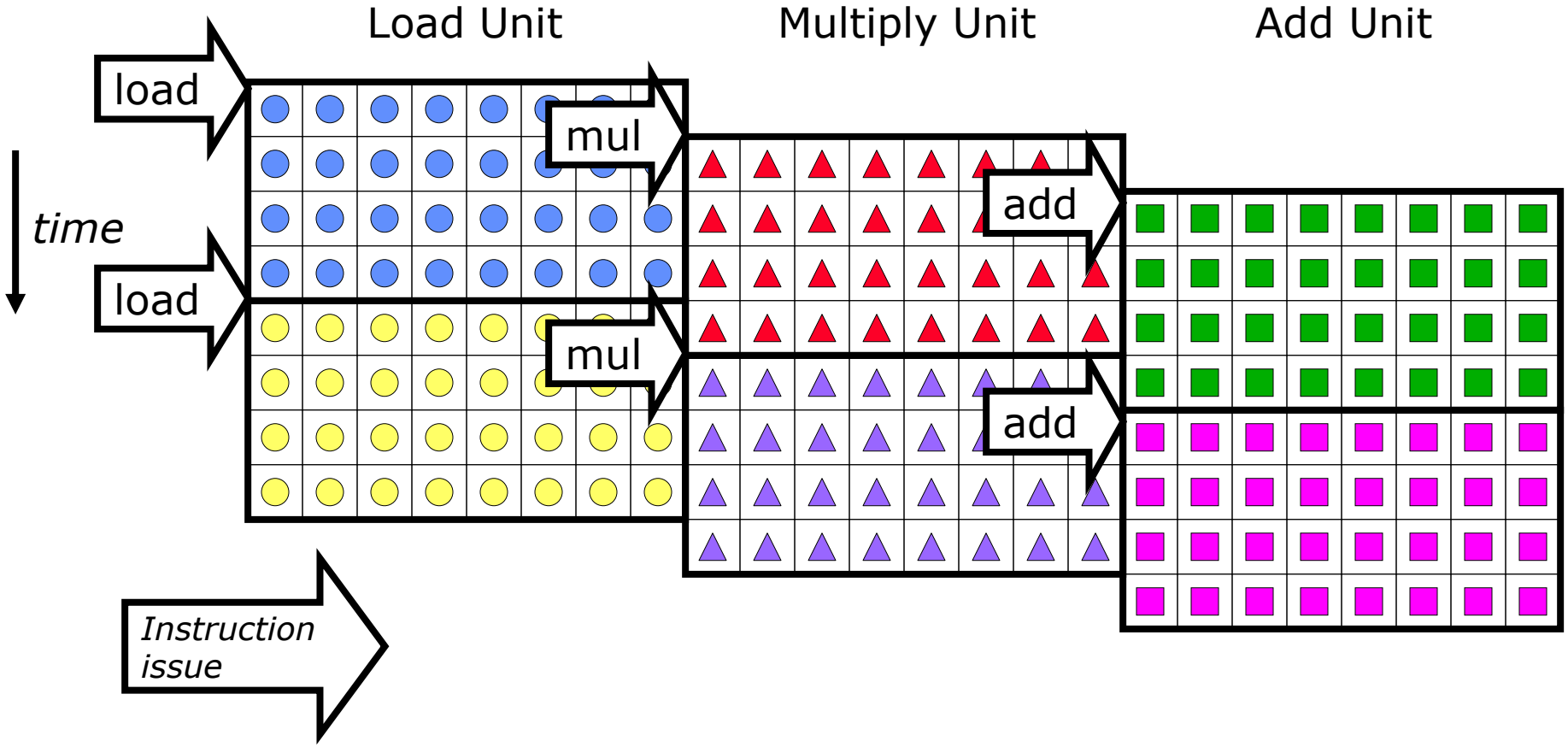


T0 Vector Microprocessor (UCB/ICSI, 1995)



Vector Instruction Parallelism

- Can overlap execution of multiple vector instructions
  - example machine has 32 elements per vector register and 8 lanes

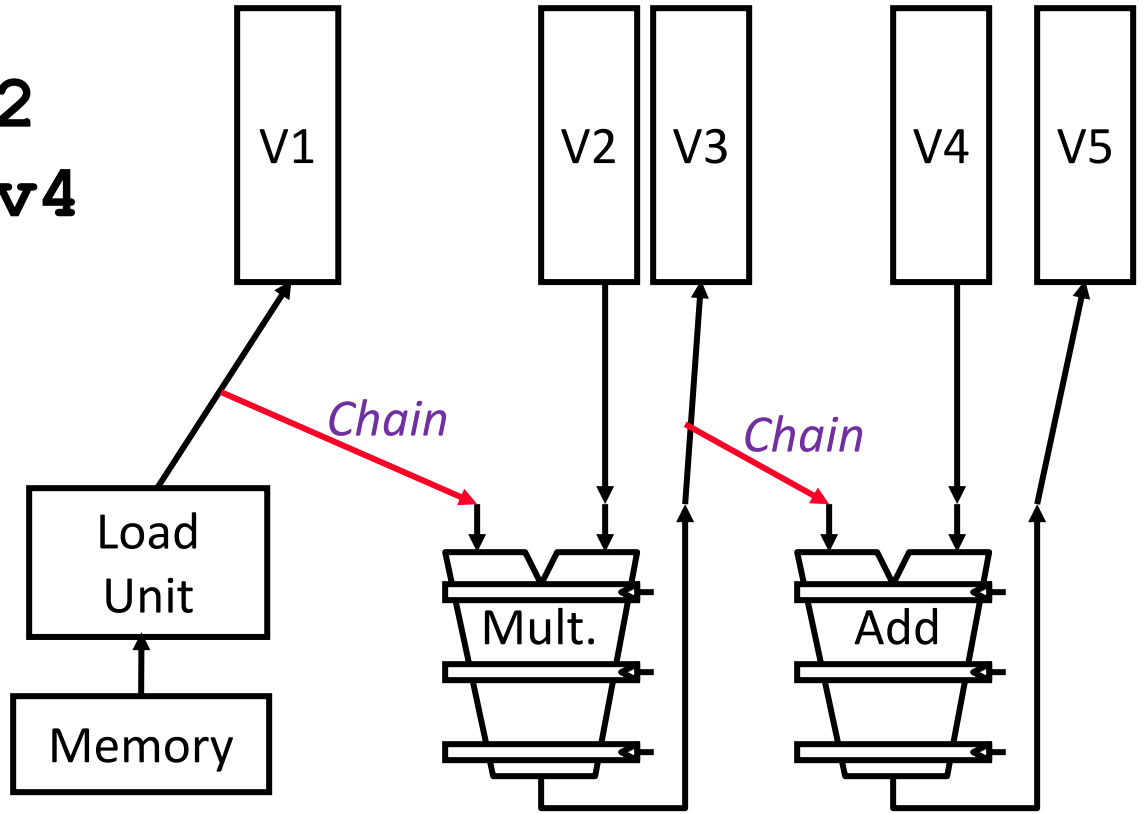


Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

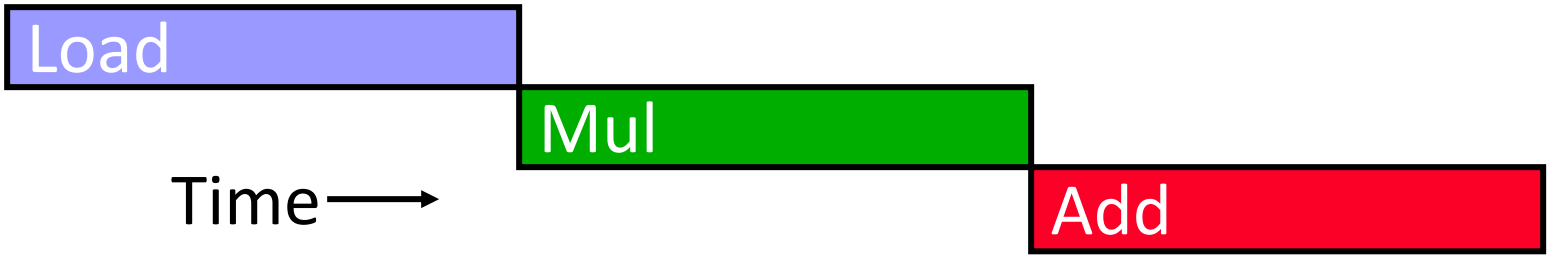
- Vector version of register bypassing
  - introduced with Cray-1

```
vld  v1
vmul v3,v1,v2
vadd v5,v3,v4
```



Vector Chaining Advantage

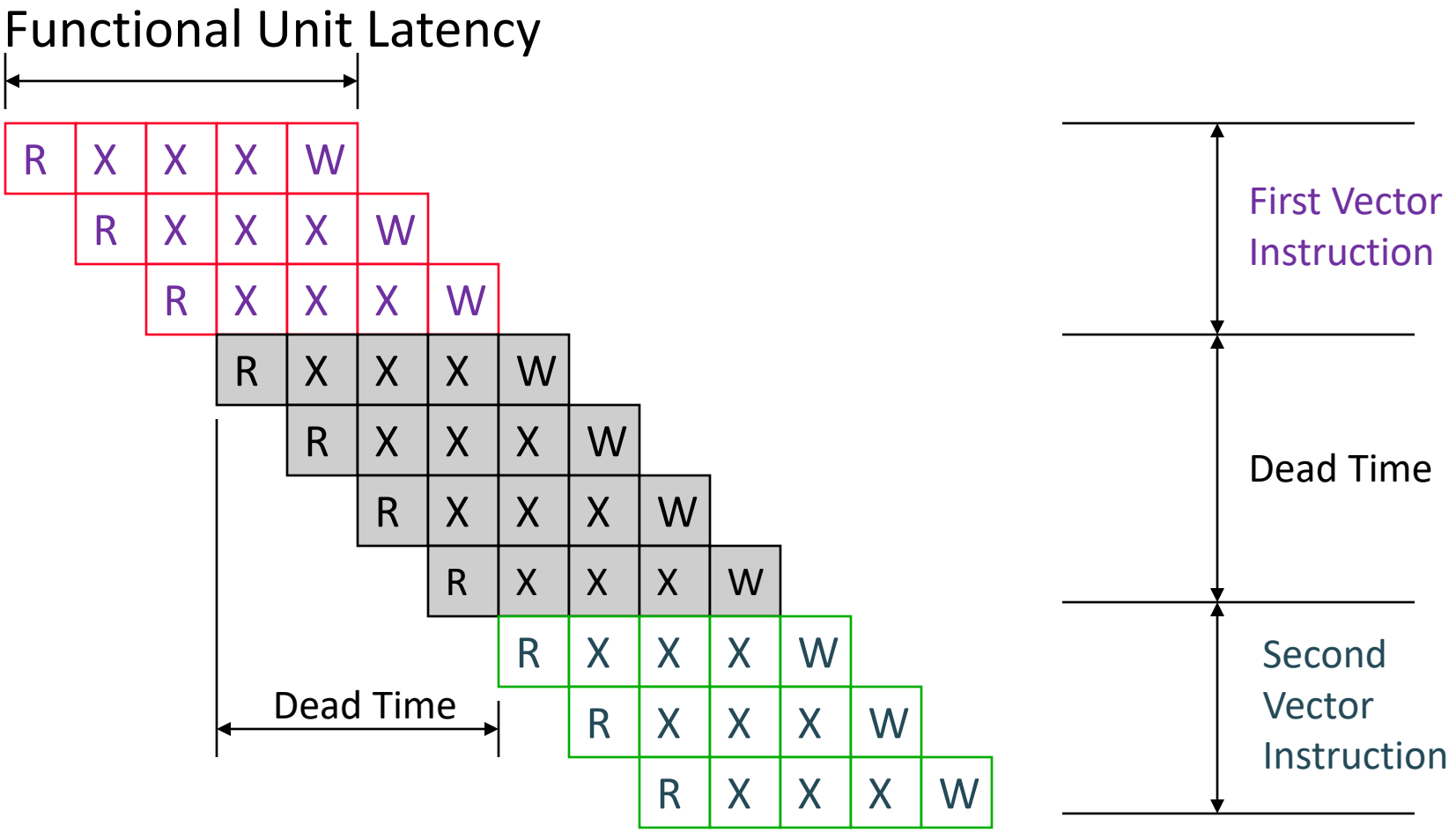
- Without chaining, must wait for last element of result to be written before starting dependent instruction



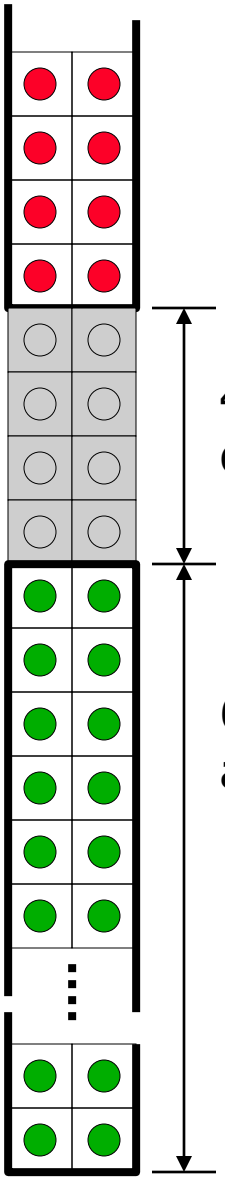
- With chaining, can start dependent instruction as soon as first result appears

Vector Startup

- Two components of vector startup penalty
  - functional unit latency (time through pipeline)
  - dead time or recovery time (time before another vector instruction can start down pipeline)



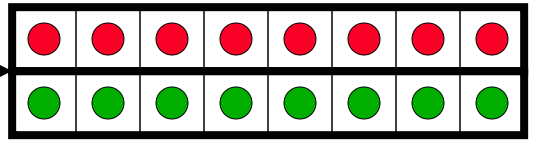
Dead Time and Short Vectors



4 cycles
dead time

64 cycles
active

No dead time



T0, Eight lanes
No dead time
100% efficiency with 8-element vectors

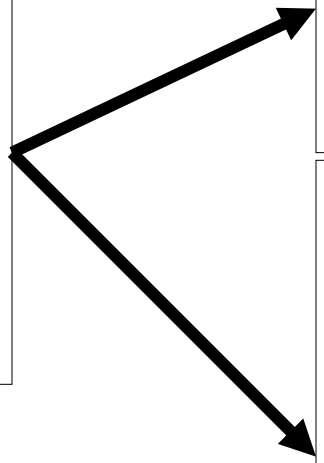
Cray C90, Two lanes
4-cycle dead time
Maximum efficiency 94% with 128-element vectors

Vector Memory-Memory versus Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
- The first vector machines, CDC Star-100 ('73) and TI ASC ('71), were memory-memory machines
- Cray-1 ('76) was first vector register machine

Example Source Code

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



Vector Memory-Memory Code

```
vadd (C),(A),(B)
vsub (D),(A),(B)
```

Vector Register Code

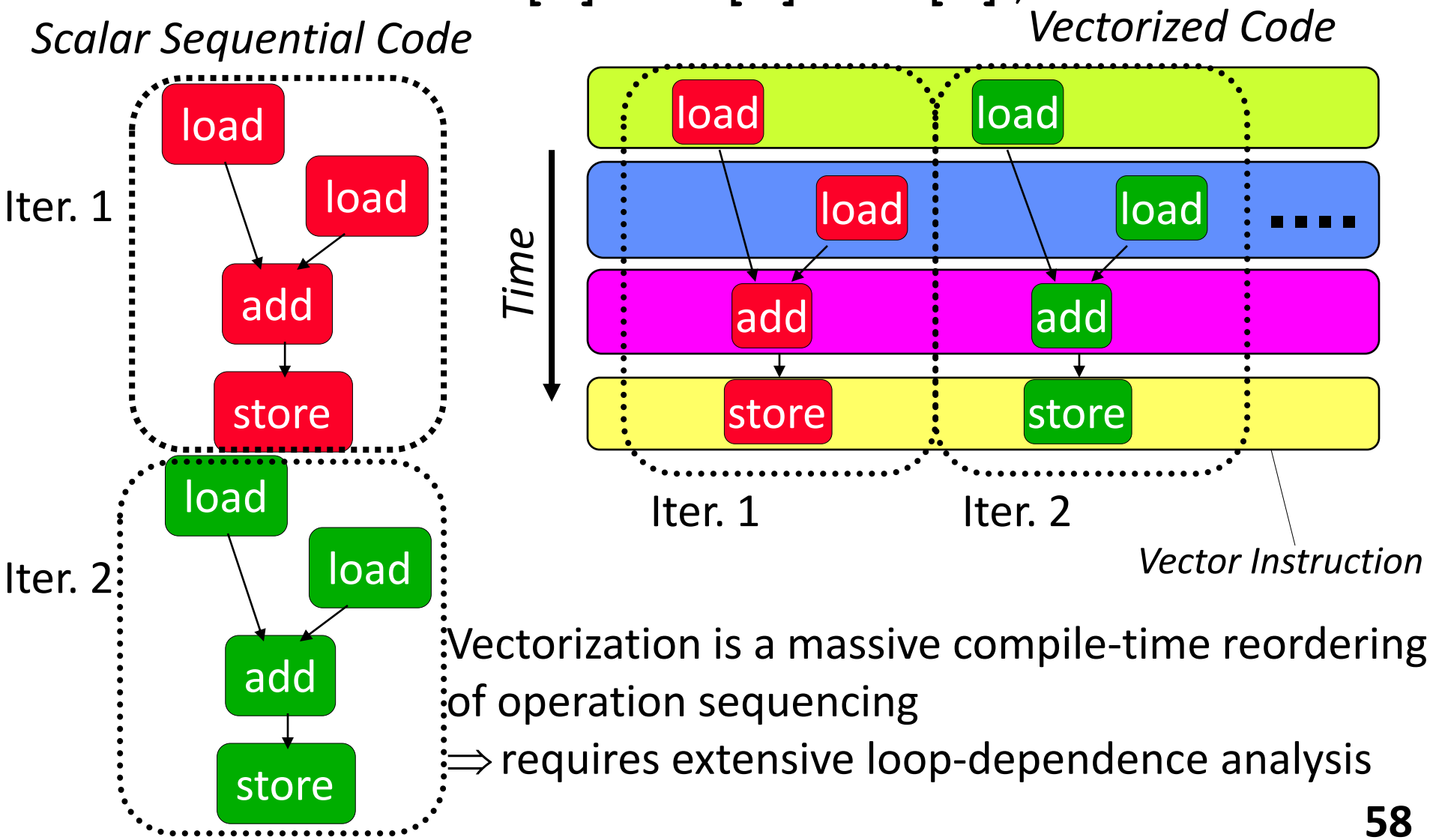
```
vld V1, (A)
vld V2, (B)
vadd V3, V1, V2
vst V3, (C)
vsub V4, V1, V2
vst V4, (D)
```

Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMAs make if difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMAs incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar breakeven point was around 2-4 elements
- Apart from CDC follow-ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures
- (we ignore vector memory-memory from now on)

Automatic Code Vectorization

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



Vectorization is a massive compile-time reordering of operation sequencing

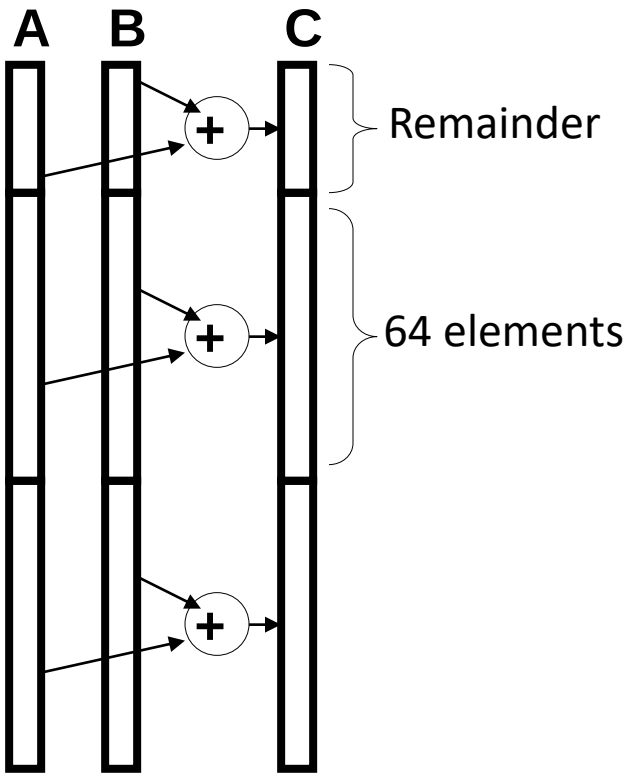
⇒ requires extensive loop-dependence analysis

Vector Stripmining

Problem: Vector registers have finite length

Solution: Break loops into pieces that fit in registers, *"Stripmining"*

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




```
      andi x1, xN, 63 # N mod 64
      vsetvl x1       # Do remainder
loop:
      vld v1, (xA)
      slli x2, x1, 3 # Multiply by 8
      add xA, xA, x2 # Bump pointer
      vld v2, (xB)
      add xB, xB, x2
      vadd v3, v1, v2
      vst v3, (xC)
      add xC, xC, x2
      sub xN, xN, x1 # Subtract elements
      li x1, 64
      vsetvl x1     # Reset full length
      bgtz xN, loop # Any more to do?
```

Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

Solution: Add vector *mask* (or *flag*) registers

– vector version of predicate registers, 1 bit per element

…and *maskable* vector instructions

– vector operation becomes bubble (“NOP”) at elements where mask bit is clear

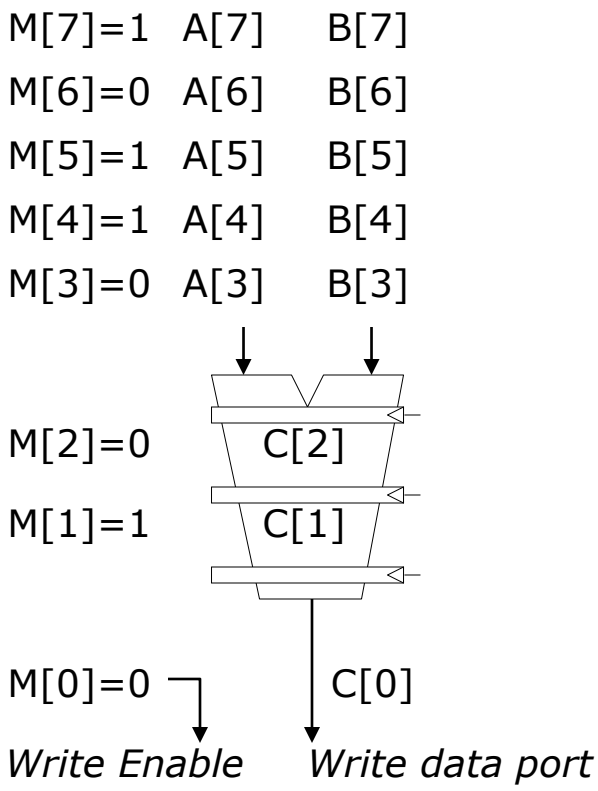
Code example:

```
cvm             # Turn on all elements
vld vA, (xA)    # Load entire A vector
vgt vA, f0      # Set bits in mask register where A>0
vld vA, (xB)    # Load B vector into A under mask
vst vA, (xA)    # Store A back to memory under mask
```

Masked Vector Instructions

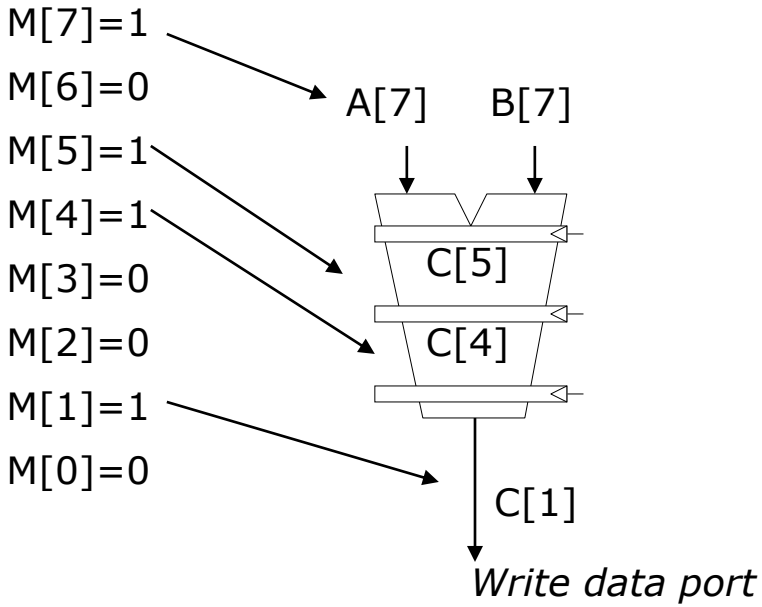
Simple Implementation

- execute all N operations, turn off result writeback according to mask



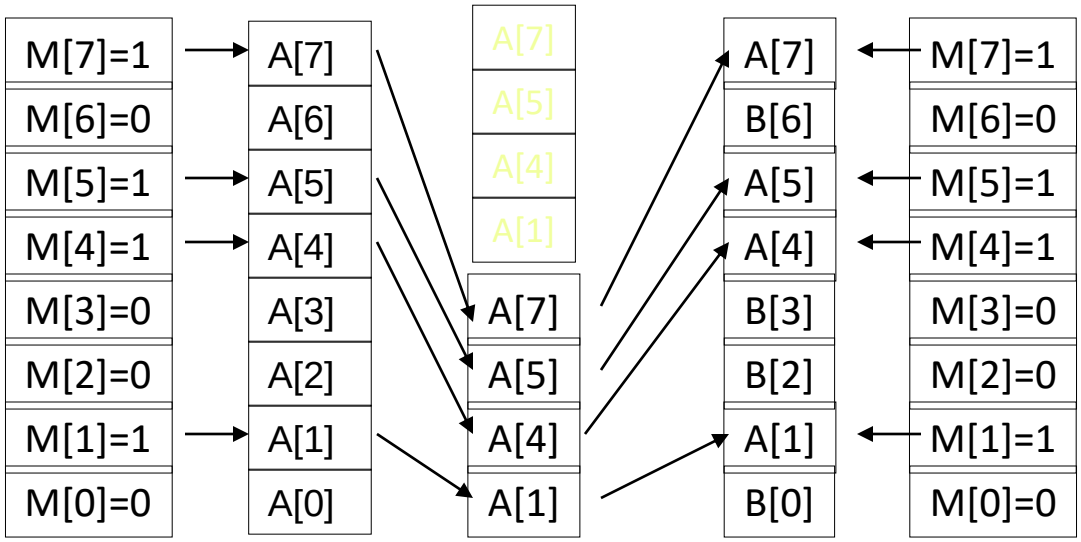
Density-Time Implementation

- scan mask vector and only execute elements with non-zero masks



Compress/Expand Operations

- Compress packs non-masked elements from one vector register contiguously at start of destination vector register
  - population count of mask vector gives packed vector length
- Expand performs inverse operation



Compress *Expand*

Used for density-time conditionals and also for general selection operations

Vector Reductions

Problem: Loop-carried dependence on reduction variables

```
sum = 0;
for (i=0; i<N; i++)
    sum += A[i];  # Loop-carried dependence on sum
```

Solution: Re-associate operations if possible, use binary tree to perform reduction

```
# Rearrange as:
sum[0:VL-1] = 0          # Vector of VL partial sums
for(i=0; i<N; i+=VL)     # Stripmine VL-sized chunks
    sum[0:VL-1] += A[i:i+VL-1]; # Vector sum
# Now have VL partial sums in one vector register
do {
    VL = VL/2;            # Halve vector length
    sum[0:VL-1] += sum[VL:2*VL-1] # Halve no. partials
} while (VL>1)
```

Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

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

Indexed load instruction (*Gather*)

```
vld vD, (xD)         # Load indices in D vector
vlx vC, (xC), vD     # Load indexed from xC base
vld vB, (xB)         # Load B vector
vadd vA,vB,vC        # Do add
vst vA, (xA)         # Store result
```

Histogram with Scatter/Gather

Histogram example:

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

Is following a correct translation?

```
vld vB, (xB)       # Load indices in B vector
vlx vA, (xA), vB   # Gather initial A values
vadd vA, vA, 1     # Increment
vsx vA, (xA), vB   # Scatter incremented values
```

Vector Memory Models

- Some vector machines have a very relaxed memory model, e.g.

```
vst v1, (x1)    # Store vector to x1
vld v2, (x1)    # Load vector from x1
```

  - No guarantee that elements of v2 will have value of elements of v1 even when store and load execute by *same* processor!

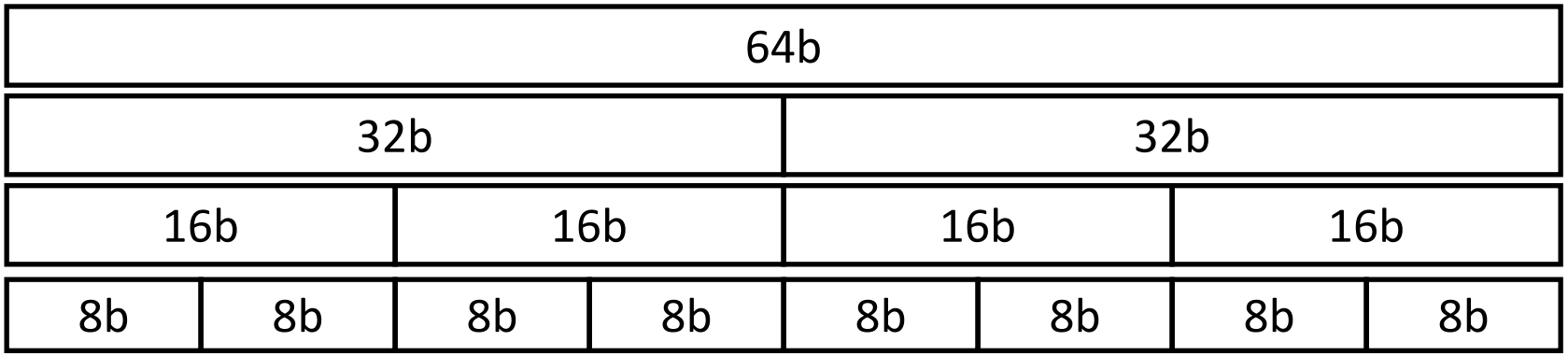
- So require explicit memory barrier or fence

```
vst v1, (x1)    # Store vector to x1
fence         # Enforce ordering s->l
vld v2, (x1)    # Load vector from x1
```

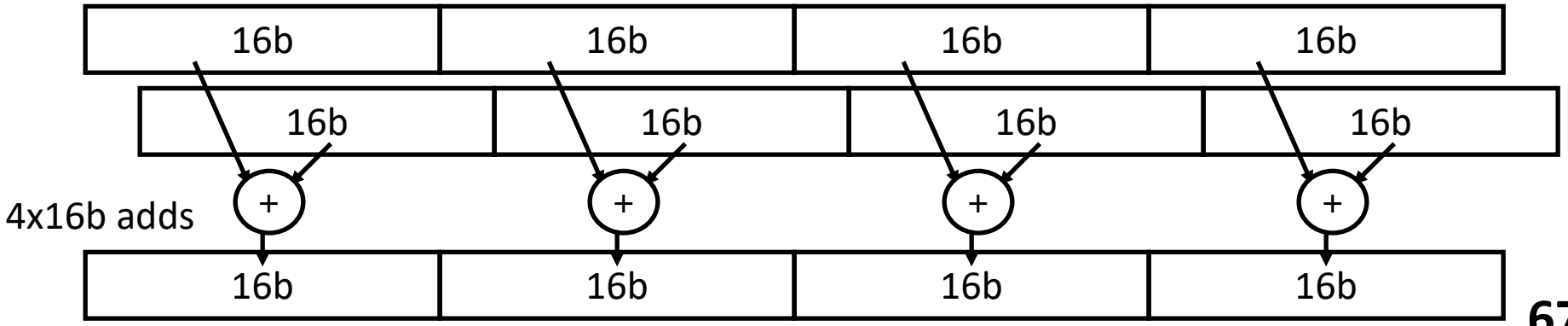
Vector machines support highly parallel memory systems (multiple lanes and multiple load and store units) with long latency (100+ clock cycles)

  - hardware coherence checks would be prohibitively expensive
  - vectorizing compiler can eliminate most dependencies

Packed SIMD Extensions



- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - Lincoln Labs TX-2 from 1957 had 36b datapath split into 2x18b or 4x9b
  - Newer designs have wider registers
    - 128b for PowerPC Altivec, Intel SSE2/3/4
    - 256b/512b for Intel AVX
- Single instruction operates on all elements within register



Packed SIMD versus Vectors

- Limited instruction set:
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-bit boundary
- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support in microprocessors
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b), gather added, scatter to follow
  - ARM Scalable Vector Extensions (SVE)

CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 09-3 GPUs
(Graphics Processing Units)

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

Types of Parallelism

- Instruction-Level Parallelism (ILP)
  - Execute independent instructions from one instruction stream in parallel (pipelining, superscalar, VLIW)
- Thread-Level Parallelism (TLP)
  - Execute independent instruction streams in parallel (multithreading, multiple cores)
- Data-Level Parallelism (DLP)
  - Execute multiple operations of the same type in parallel (vector/SIMD execution)

- Which is easiest to program?
- Which is most flexible form of parallelism?
  - i.e., can be used in more situations
- Which is most efficient?
  - i.e., greatest tasks/second/area, lowest energy/task

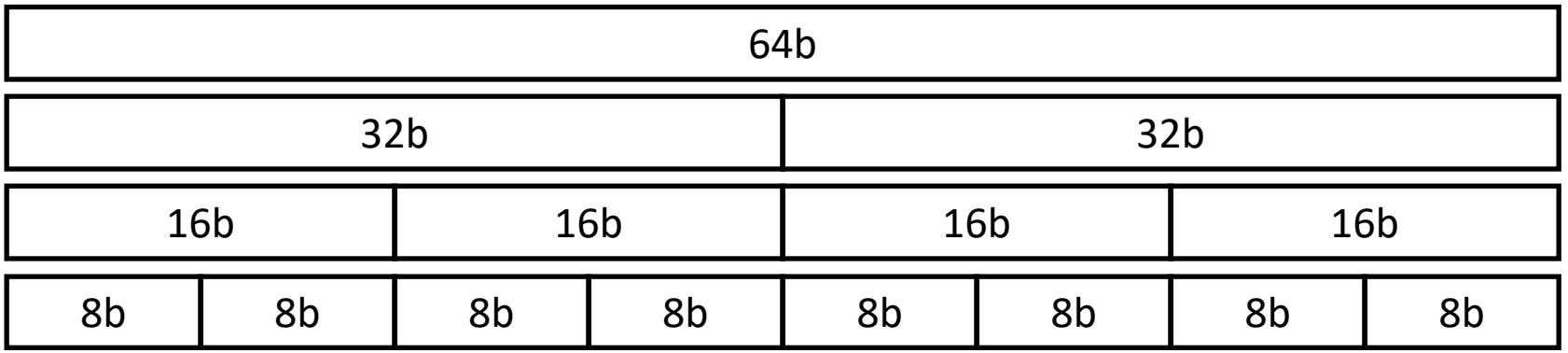
Resurgence of DLP

- Convergence of application demands and technology constraints drives architecture choice

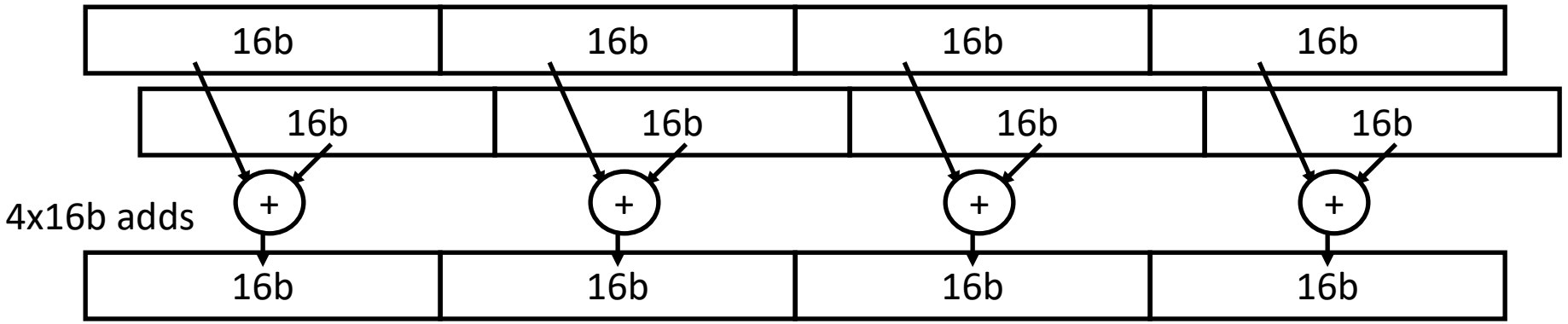
- New applications, such as graphics, machine vision, speech recognition, machine learning, etc. all require large numerical computations that are often trivially data parallel

- SIMD-based architectures (vector-SIMD, subword-SIMD, SIMT/GPUs) are most efficient way to execute these algorithms

Packed SIMD Extensions



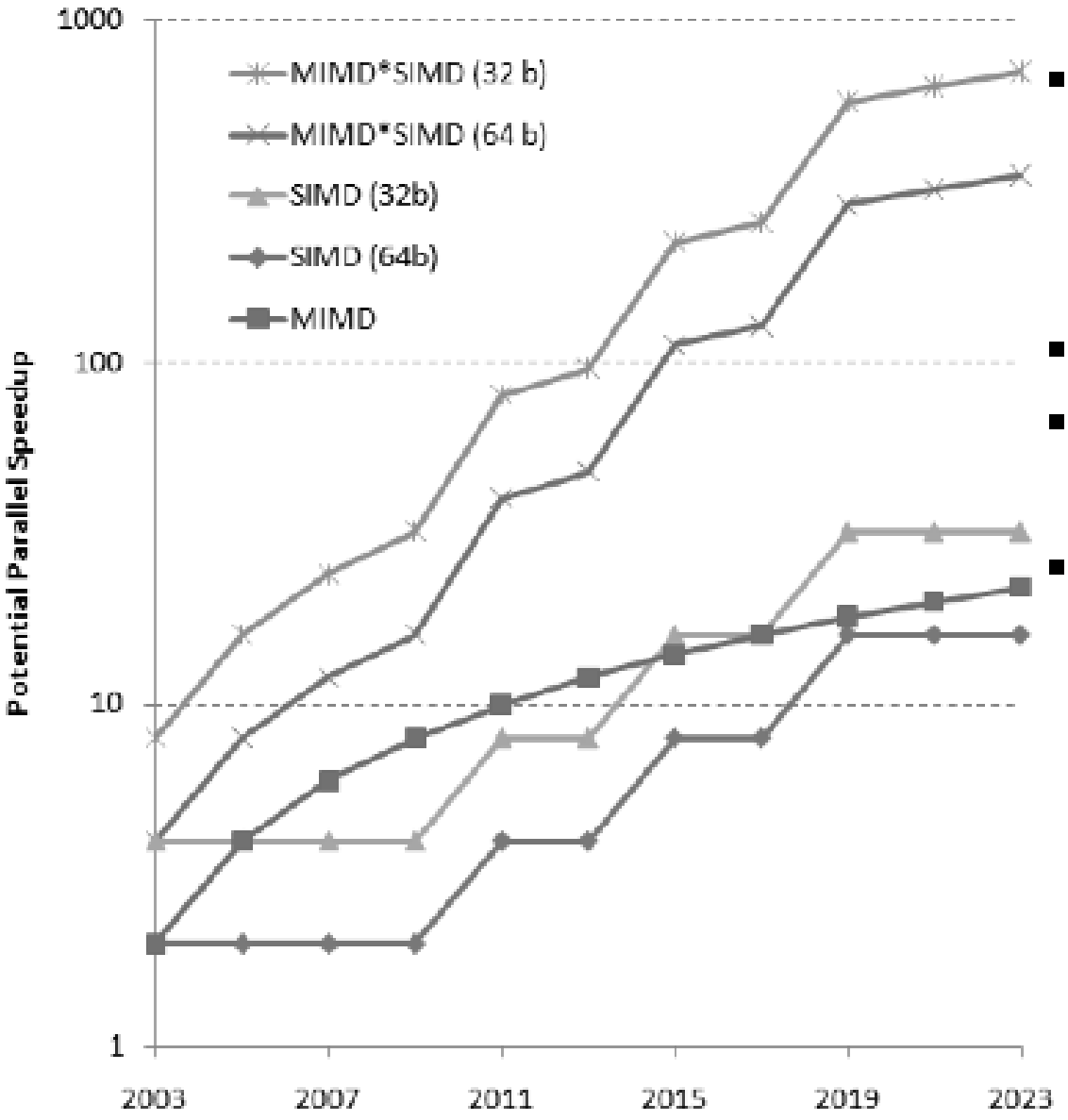
- Short vectors added to existing microprocessors ISAs, for multimedia
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - Lincoln Labs TX-2 from 1957 had 36b datapath split into 2x18b or 4x9b
  - Newer designs have wider registers
    - 128b for PowerPC Altivec, Intel SSE2/3/4
    - 256b for Intel AVX
- Single instruction operates on all elements within register



Multimedia Extensions versus Vectors

- Limited instruction set
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be naturally aligned to whole register width (e.g., 64 or 128-bit)
- Limited vector register length
  - requires superscalar issue to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support in microprocessors
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b) , adding scatter/gather
  - New ARM SVE/MVE vector ISA closer to traditional vector designs

DLP important for conventional CPUs



- Prediction for x86 processors, from Hennessy & Patterson, 5th edition
  - ***Note: Educated guess, not Intel product plans!***
- TLP: 2+ cores / 2 years
- DLP: 2x width / 4 years
- DLP will account for more mainstream parallelism growth than TLP in next decade.
  - SIMD –single-instruction multiple-data (DLP)
  - MIMD- multiple-instruction multiple-data (TLP)

Graphics Processing Units (GPUs)

- Original GPUs were dedicated fixed-function devices for generating 3D graphics (mid-late 1990s) including high-performance floating-point units
  - Provide workstation-like graphics for PCs
  - User could configure graphics pipeline, but not really program it
- Over time, more programmability added (2001-2005)
  - E.g., New language Cg for writing small programs run on each vertex or each pixel, also Windows DirectX variants
  - Massively parallel (millions of vertices or pixels per frame) but very constrained programming model
- Some users noticed they could do general-purpose computation by mapping input and output data to images, and computation to vertex and pixel shading computations
  - Incredibly difficult programming model as had to use graphics pipeline model for general computation

General-Purpose GPUs (GP-GPUs)

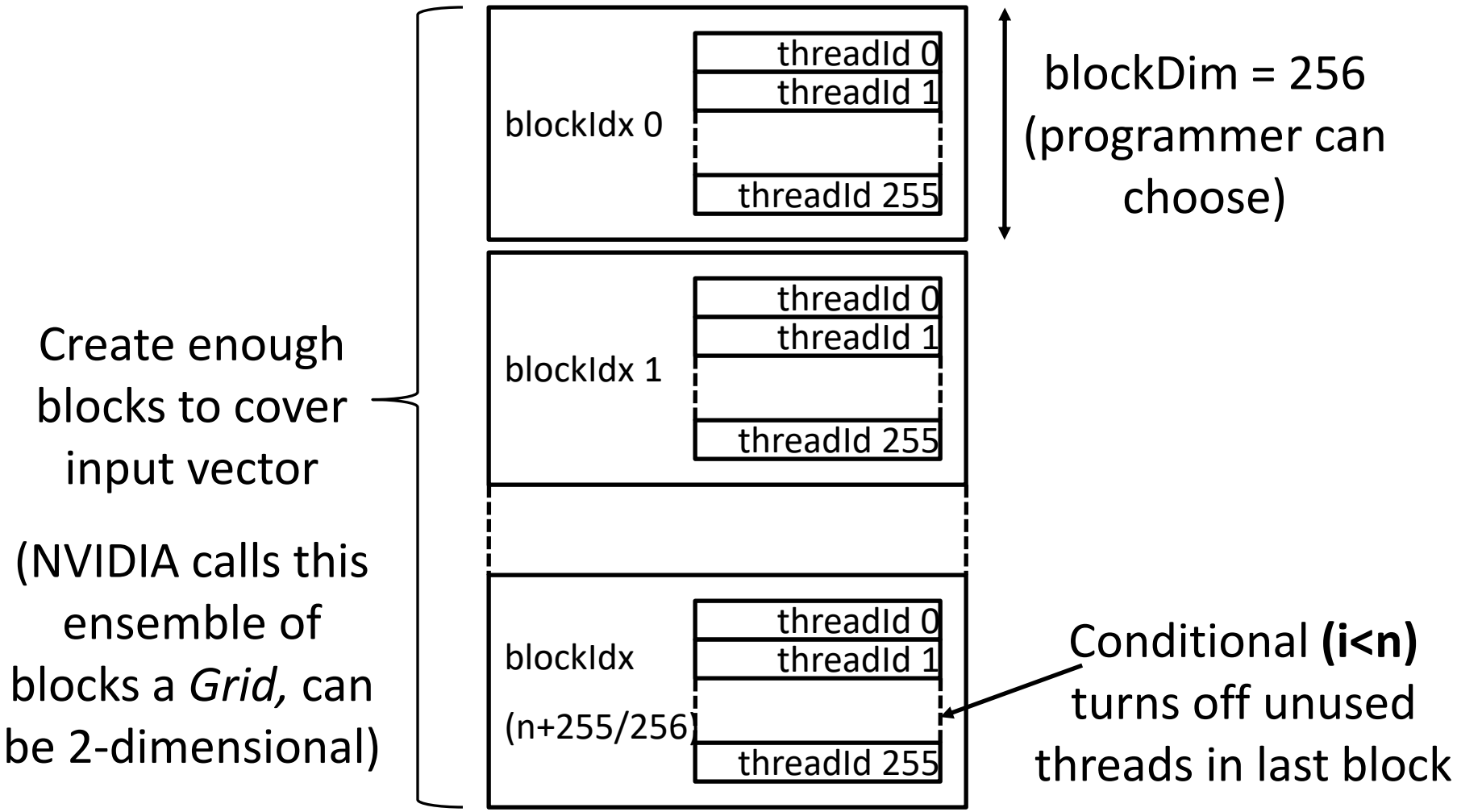
- In 2006, Nvidia introduced GeForce 8800 GPU, which supported a new programming language CUDA (in 2007)
  - "Compute Unified Device Architecture"
  - Subsequently, broader industry pushing for OpenCL, a vendor-neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution
- This lecture has a simplified version of Nvidia CUDA-style model and only considers GPU execution for computational kernels, not graphics
  - Would need whole other course to describe graphics processing

Simplified CUDA Programming Model

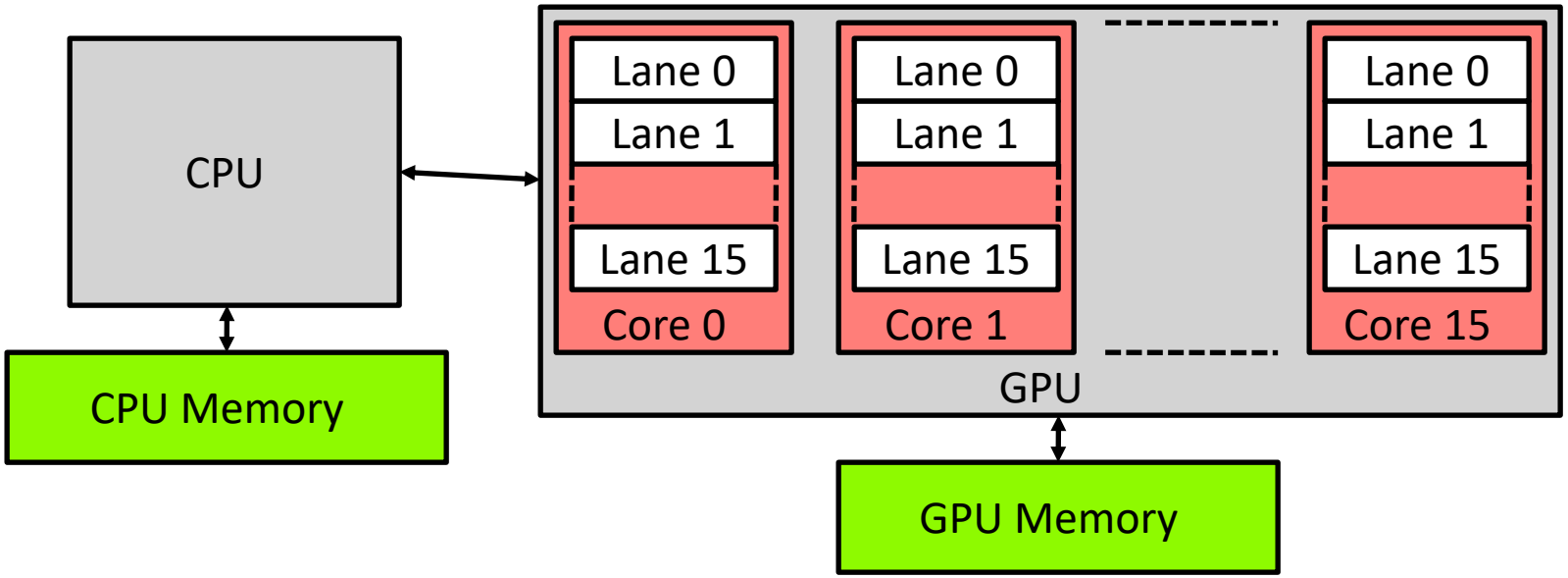
- Computation performed by a very large number of independent small scalar threads (*CUDA threads* or *microthreads*) grouped into *thread blocks.*

```
// C version of DAXPY loop.
void daxpy(int n, double a, double*x, double*y)
{ for (int i=0; i<n; i++)
      y[i] = a*x[i] + y[i]; }

// CUDA version.
__host__  // Piece run on host processor.
int nblocks = (n+255)/256; //256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);

__device__  // Piece run on GP-GPU.
void daxpy(int n, double a, double*x, double*y)
{ int i = blockIdx.x*blockDim.x + threadId.x;
  if (i<n) y[i]=a*x[i]+y[i]; }
```

Programmer's View of Execution



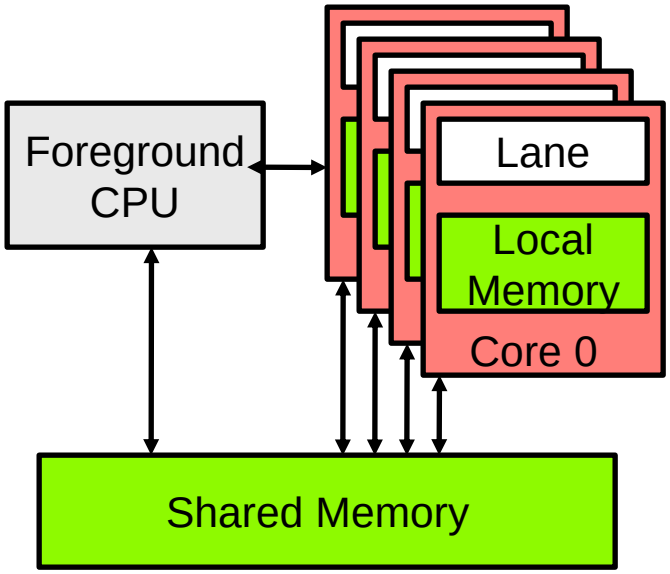
Hardware Execution Model

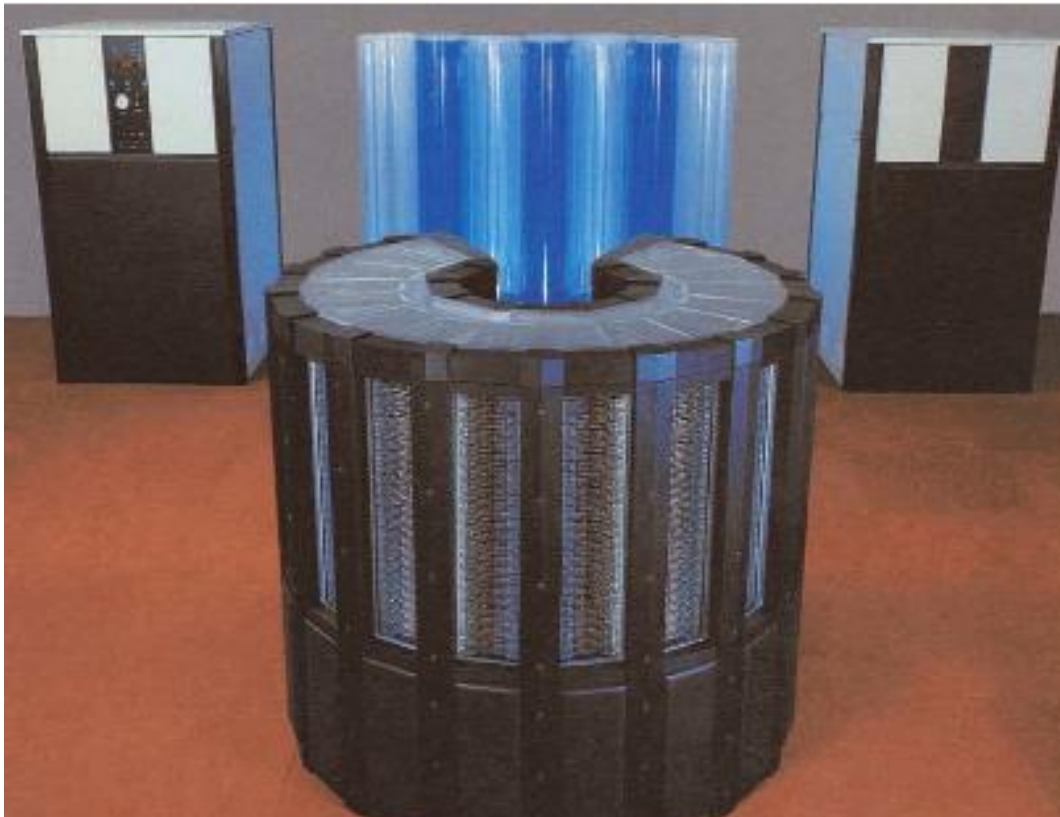


- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor
  - some adding “scalar coprocessors” now
- CPU sends whole “grid” over to GPU, which distributes thread blocks among cores (each thread block executes on one core)
  - Programmer unaware of number of cores

Historical Retrospective, Cray-2 (1985)

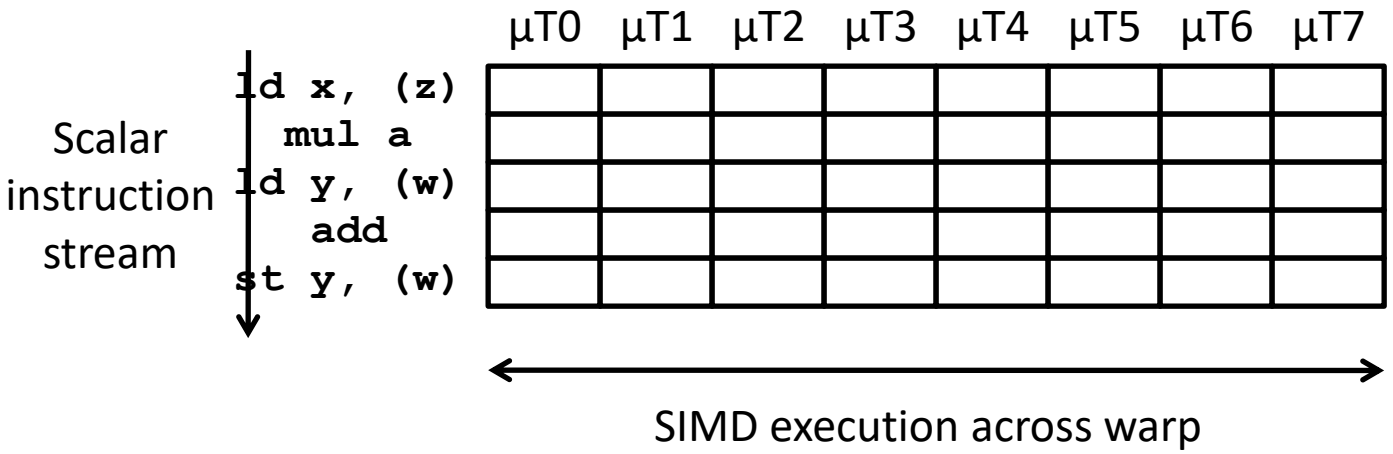
- 243MHz ECL logic
- 2GB DRAM main memory (128 banks of 16MB each)
  - Bank busy time 57 clocks!
- Local memory of 128KB/core
- 1 foreground + 4 background vector processors





“Single Instruction, Multiple Thread” (SIMT)

- GPUs use a SIMT model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (NVIDIA groups 32 CUDA threads into a *warp*)

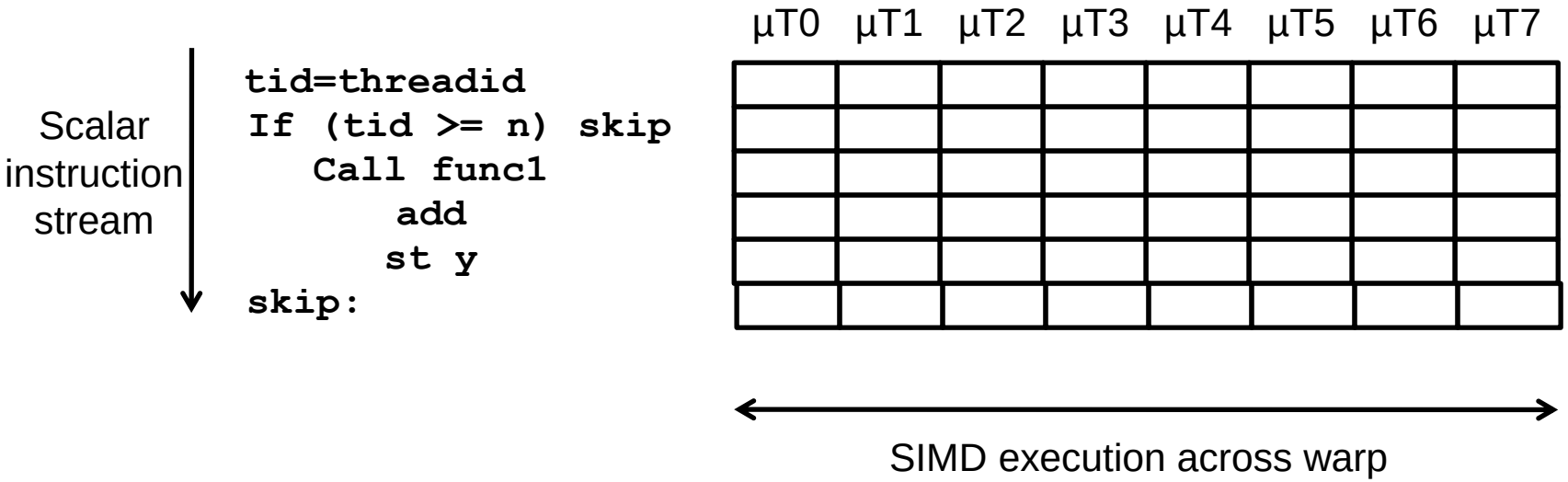


Implications of SIMT Model

- All “vector” loads and stores are scatter-gather, as individual µthreads perform scalar loads and stores
  - GPU adds hardware to dynamically coalesce individual µthread loads and stores to mimic vector loads and stores
- Every µthread has to perform stripmining calculations redundantly (“am I active?”) as there is no scalar processor equivalent

Conditionals in SIMT model

- Simple if-then-else are compiled into predicated execution, equivalent to vector masking
- More complex control flow compiled into branches
- How to execute a vector of branches? Vector function calls?



Branch Divergence

- Hardware tracks which µthreads take or don't take branch
- If all go the same way, then keep going in SIMD fashion
- If not, create mask vector indicating taken/not-taken
- Keep executing not-taken path under mask, push taken branch PC+mask onto a hardware stack and execute later
- When can execution of µthreads in warp reconverge?

NVIDIA Instruction Set Arch.

- ISA is an abstraction of the hardware instruction set
  - "Parallel Thread Execution (PTX)"
    - opcode.type d,a,b,c;
  - Uses virtual registers
  - Translation to machine code is performed in software
  - Example:

```
shl.s32         R8, blockIdx, 9         ; Thread Block ID * Block size (512 or 29)
add.s32         R8, R8, threadIdx       ; R8 = i = my CUDA thread ID
ld.global.f64   RD0, [X+R8]             ; RD0 = X[i]
ld.global.f64   RD2, [Y+R8]             ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4       ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2       ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0   ; Y[i] = sum (X[i]*a + Y[i])
```

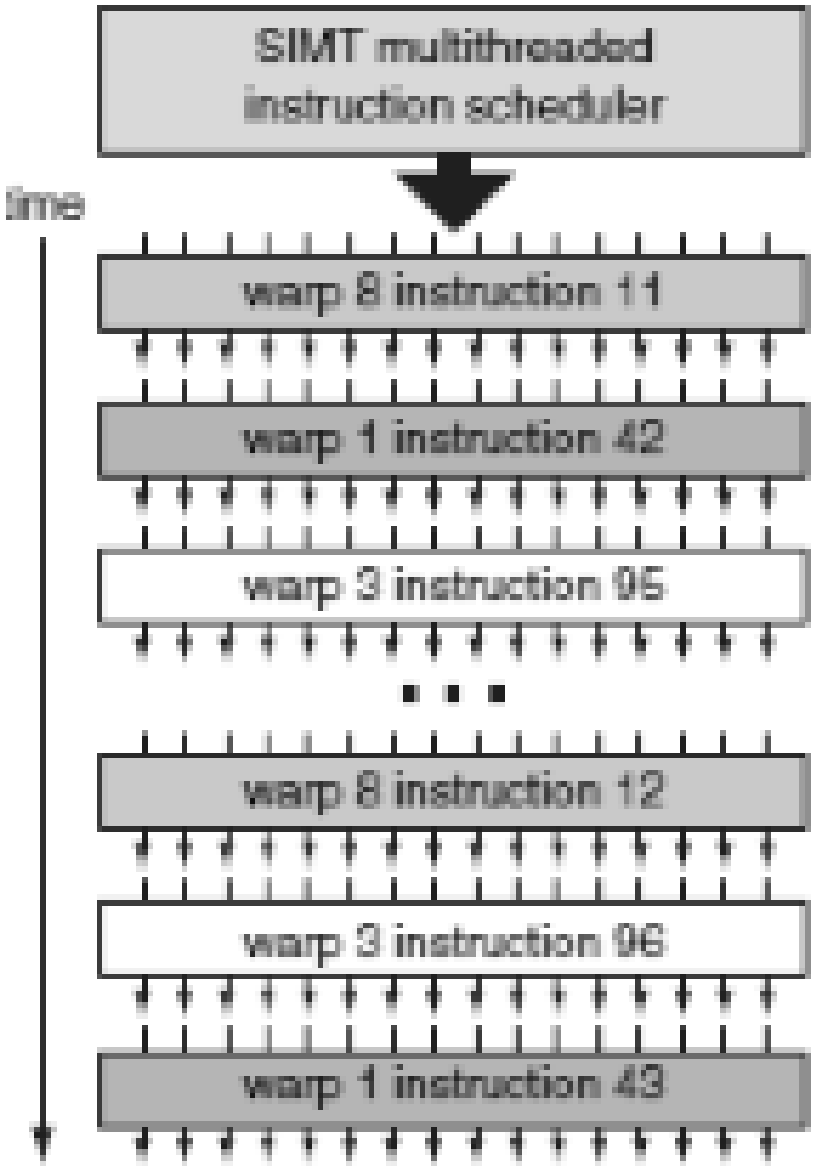
Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

Example

```
    if (X[i] != 0)
            X[i] = X[i] – Y[i];
    else X[i] = Z[i];

    ld.global.f64       RD0, [X+R8]                 ; RD0 = X[i]
    setp.neq.s32        P1, RD0, #0                 ; P1 is predicate register 1
    @!P1, bra           ELSE1, *Push                ; Push old mask, set new mask bits
                                                    ; if P1 false, go to ELSE1
    ld.global.f64       RD2, [Y+R8]                 ; RD2 = Y[i]
    sub.f64  RD0, RD0, RD2                  ; Difference in RD0
    st.global.f64       [X+R8], RD0                 ; X[i] = RD0
    @P1, bra            ENDIF1, *Comp               ; complement mask bits
                                                    ; if P1 true, go to ENDIF1
ELSE1:                  ld.global.f64 RD0, [Z+R8]   ; RD0 = Z[i]
                        st.global.f64 [X+R8], RD0   ; X[i] = RD0
ENDIF1:     <next instruction>, *Pop        ; pop to restore old mask
```

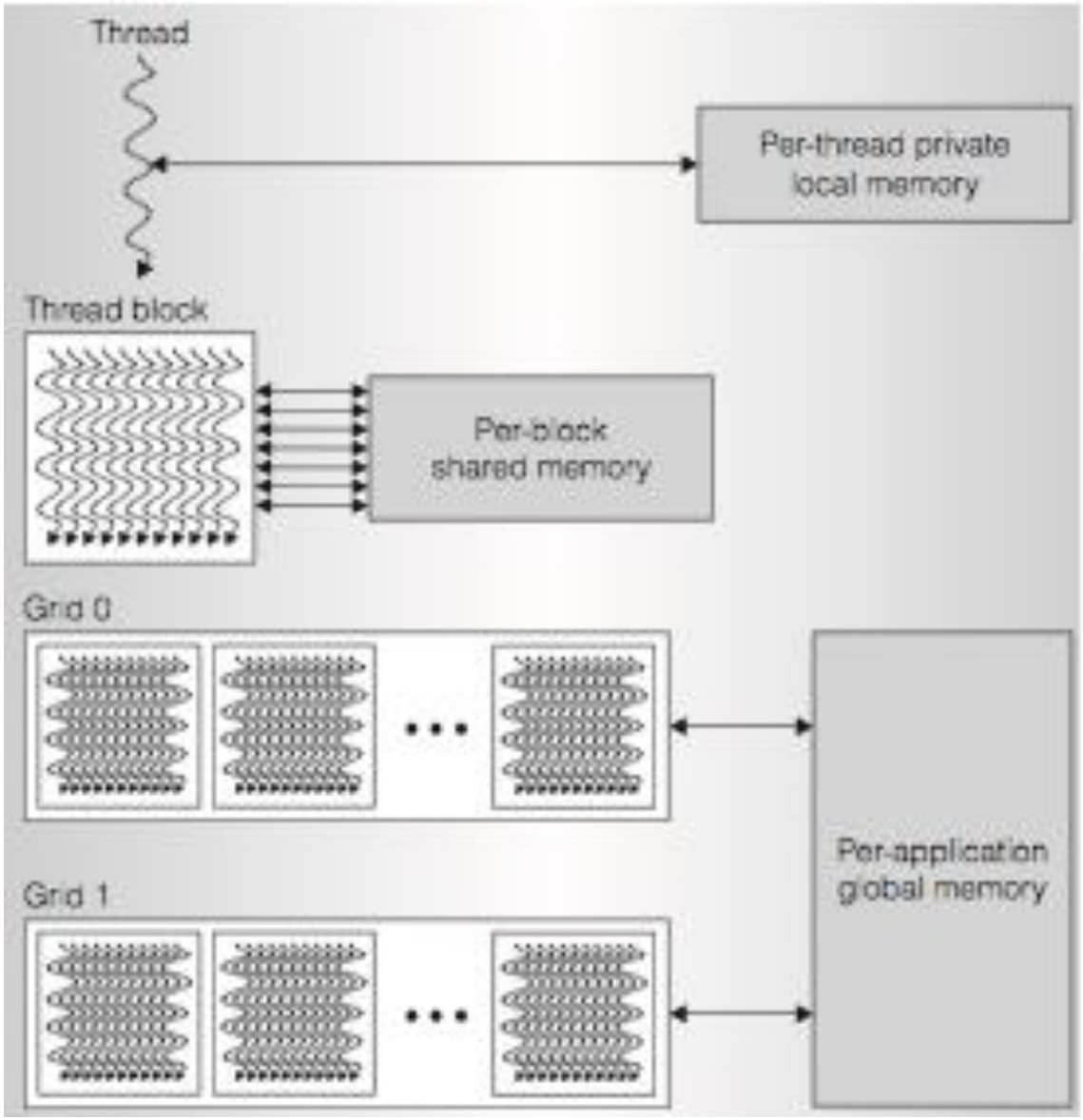
Warps are multithreaded on core



[Nvidia, 2010]

- One warp of 32 µthreads is a single thread in the hardware
- Multiple warp threads are interleaved in execution on a single core to hide latencies (memory and functional unit)
- A single thread block can contain multiple warps (up to 512 µT max in CUDA), all mapped to single core
- Can have multiple blocks executing on one core

GPU Memory Hierarchy



[Nvidia, 2010]

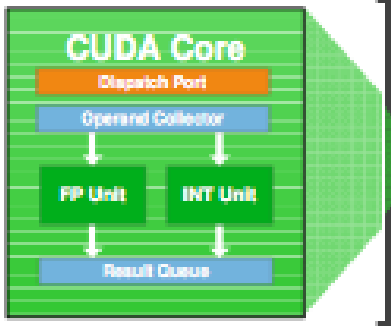
SIMT

- Illusion of many independent threads
- But for efficiency, programmer must try and keep µthreads aligned in a SIMD fashion
  - Try and do unit-stride loads and store so memory coalescing kicks in
  - Avoid branch divergence so most instruction slots execute useful work and are not masked off

Nvidia Fermi GF100 GPU

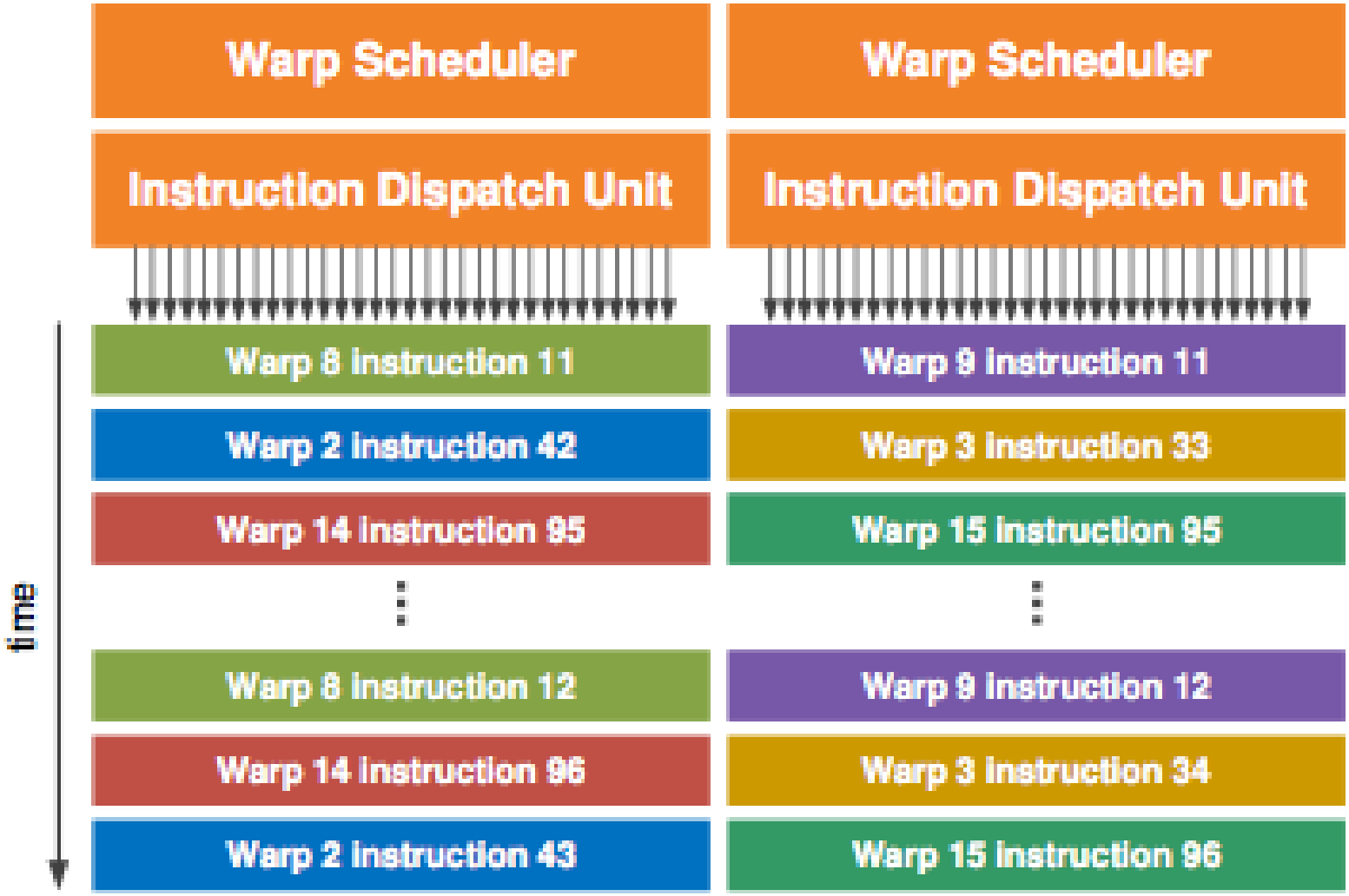
[Nvidia, 2010]

Fermi "Streaming Multiprocessor" Core



NVIDIA Pascal Multithreaded GPU Core

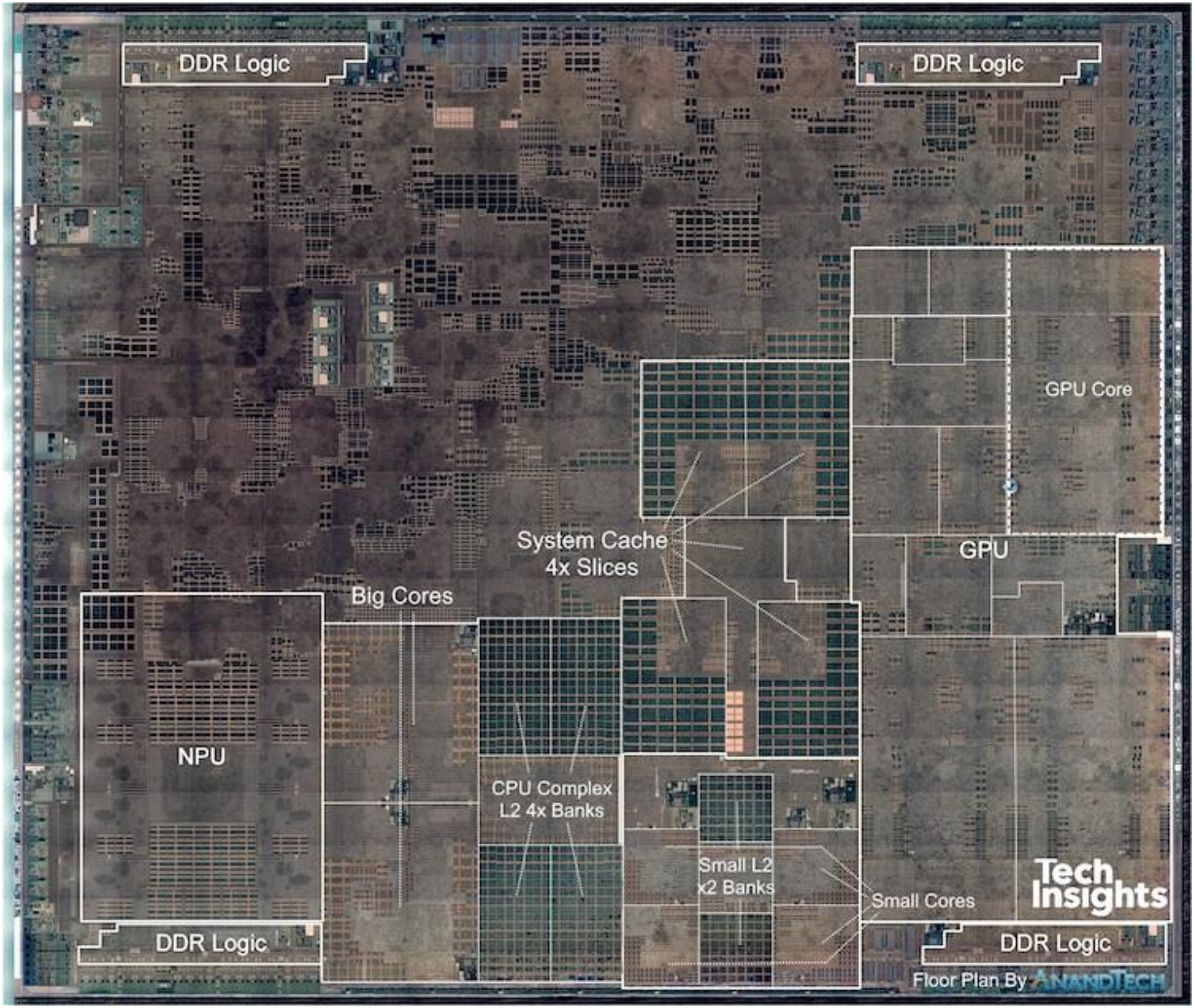
Fermi Dual-Issue Warp Scheduler



Important of Machine Learning for GPUs

NVIDIA stock price 40x in 9 years (since deep learning became important)

Apple A12 Processor (2018)



- 83.27mm^2
- 7nm technology

[Source: Tech Insights, AnandTech]