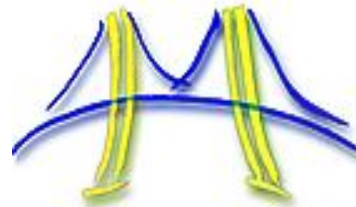


PARLab Parallel Boot Camp



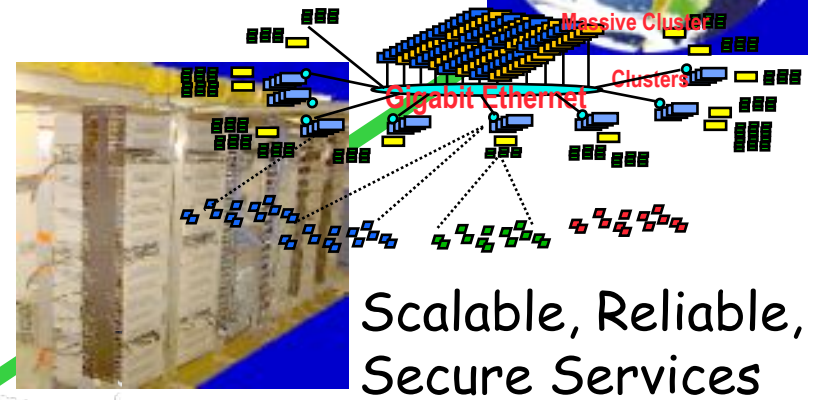
Introduction to
Parallel Computer Architecture
9:30am-12pm

John Kubiawicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

Societal Scale Information Systems

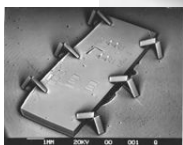


- The world *is* a large parallel system already
 - Microprocessors in everything
 - Vast infrastructure behind this
 - People who say that parallel computing never took off have not been watching
- So: why are people suddenly so excited about parallelism?
 - *Because Parallelism is being forced upon the lowest level*



Databases
 Information Collection
 Remote Storage
 Online Games
 Commerce

...

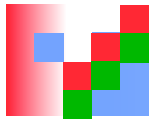


MEMS for
 Sensor Nets
 8/19/09

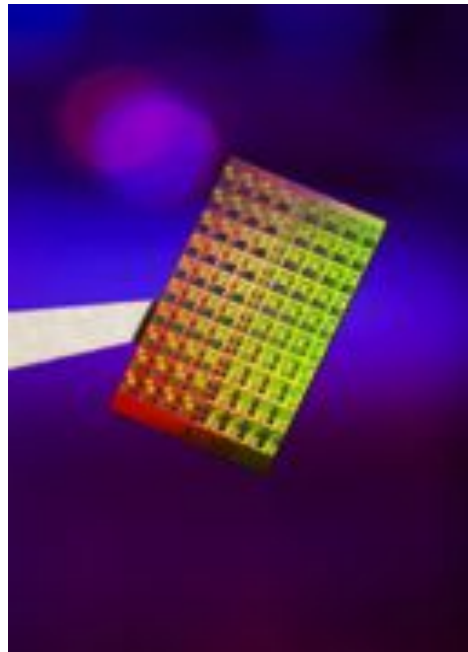
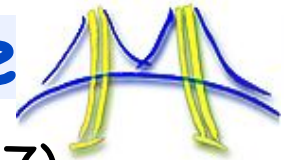
Building & Using
 Sensor Nets

John Kubiawicz

Parallel Architecture: 2



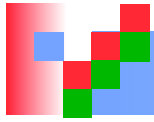
ManyCore Chips: The future is here



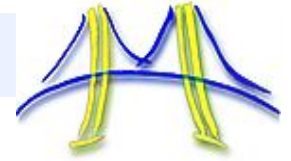
- Intel 80-core multicore chip (Feb 2007)
 - 80 simple cores
 - Two floating point engines /core
 - Mesh-like "network-on-a-chip"
 - 100 million transistors
 - 65nm feature size

Frequency	Voltage	Power	Bandwidth	Performance
3.16 GHz Teraflops	0.95 V	62W	1.62 Terabits/s	1.01
5.1 GHz Teraflops	1.2 V	175W	2.61 Terabits/s	1.63
5.7 GHz Teraflops	1.35 V	265W	2.92 Terabits/s	1.81

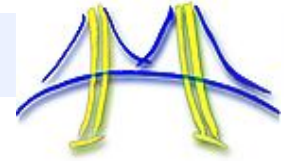
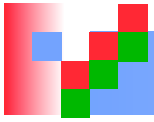
- "ManyCore" refers to many processors/chip
 - 64? 128? Hard to say exact boundary
- How to program these?
 - Use 2 CPUs for video/audio
 - Use 1 for word processor, 1 for browser
 - 76 for virus checking???
- Something new is clearly needed here...



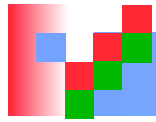
Outline of Today's Lesson



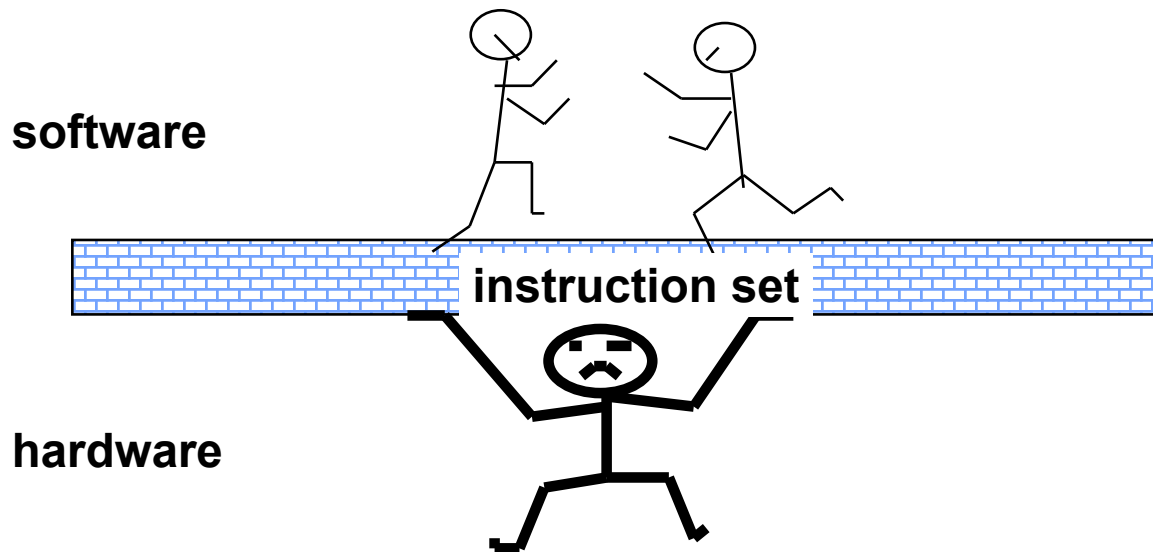
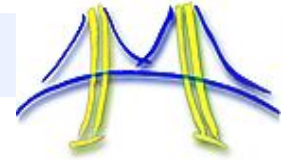
- **Goals:**
 - Pick up some common terminology/concepts for later in the course
- **Uniprocessor Parallelism**
 - Pipelining, Superscalar, Out-of-order execution
 - Vector Processing/SIMD
 - Multithreading
 - Uniprocessor Memory Systems
- **Parallel Computer Architecture**
 - Programming Models
 - Shared Memory/Synchronization primitives
 - Message Passing
- **Actual Parallel Machines/Multicore chips**



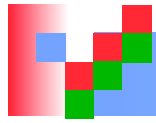
Computer Architecture



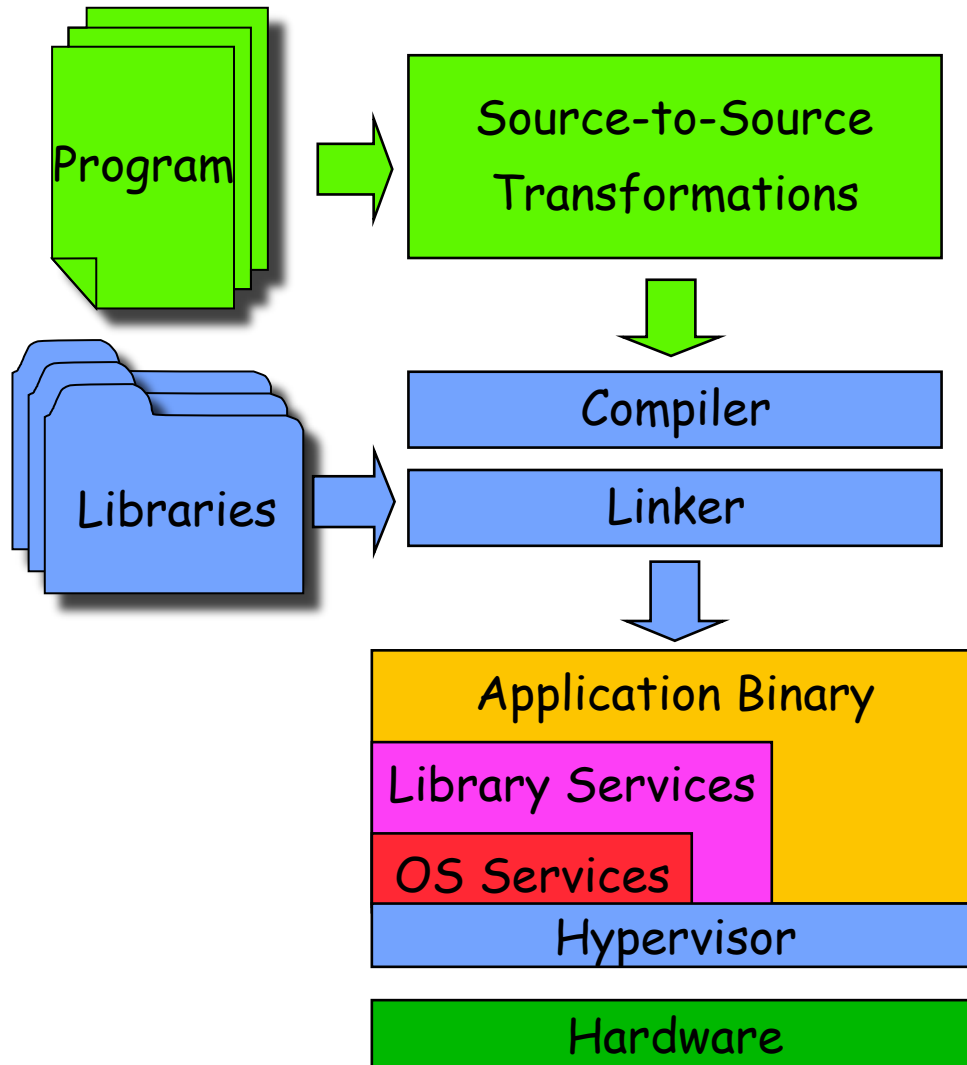
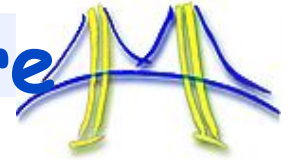
Hardware/Software Interface



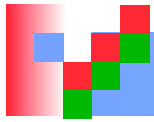
- Properties of a good abstraction
 - Lasts through many generations (portability)
 - Used in many different ways (generality)
 - Provides **convenient** functionality to higher levels
 - Permits an **efficient** implementation at lower levels
- But: Lessons of RISC
 - What is important is the combination view presented to *programmer*
NOT necessarily the compiler/OS
 - Hardware should *never* be optimized in the absence of the environment



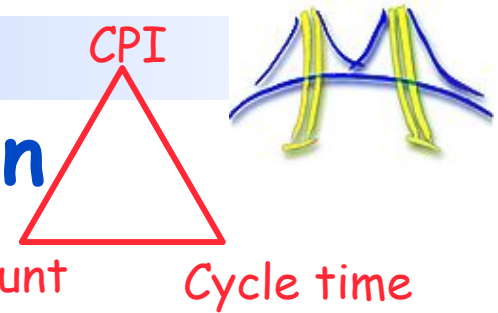
Execution is *not* just about hardware



- The VAX fallacy
 - Produce one instruction for every high-level concept
 - Absurdity: Polynomial Multiply
 - » Single hardware instruction
 - » But Why? Is this really faster???
- RISC Philosophy
 - Full System Design
 - Hardware mechanisms viewed in *context* of complete system
 - Cross-boundary optimization
- Modern programmer does not see assembly language
 - Many do not even see "low-level" languages like "C".

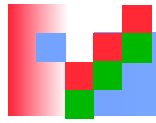


Not Fooling Yourself: Processor performance equation

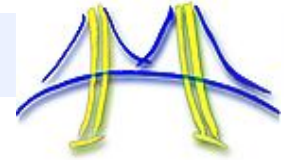


$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Inst Count	CPI	Clock Rate
Program	X		
Compiler	X	(X)	
Inst. Set.	X	X	
Organization	X		X
Technology			X



Not Fooling Yourself: Amdahl's Law

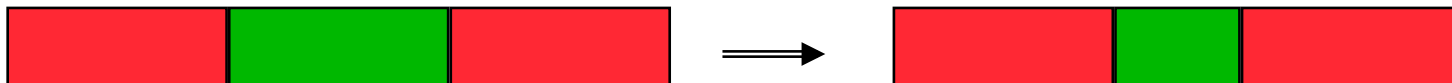


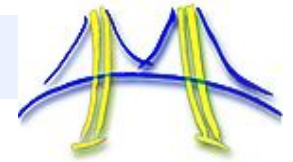
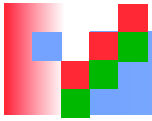
$$\text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times \left[(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right]$$

$$\text{Speedup}_{\text{overall}} = \frac{\text{ExTime}_{\text{old}}}{\text{ExTime}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

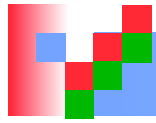
Best you could ever hope to do:

$$\text{Speedup}_{\text{maximum}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}})}$$

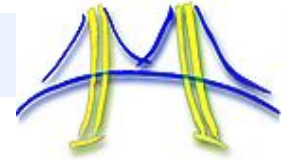




Uniprocessor Parallelism

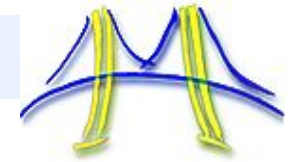


Parallelism is Everywhere



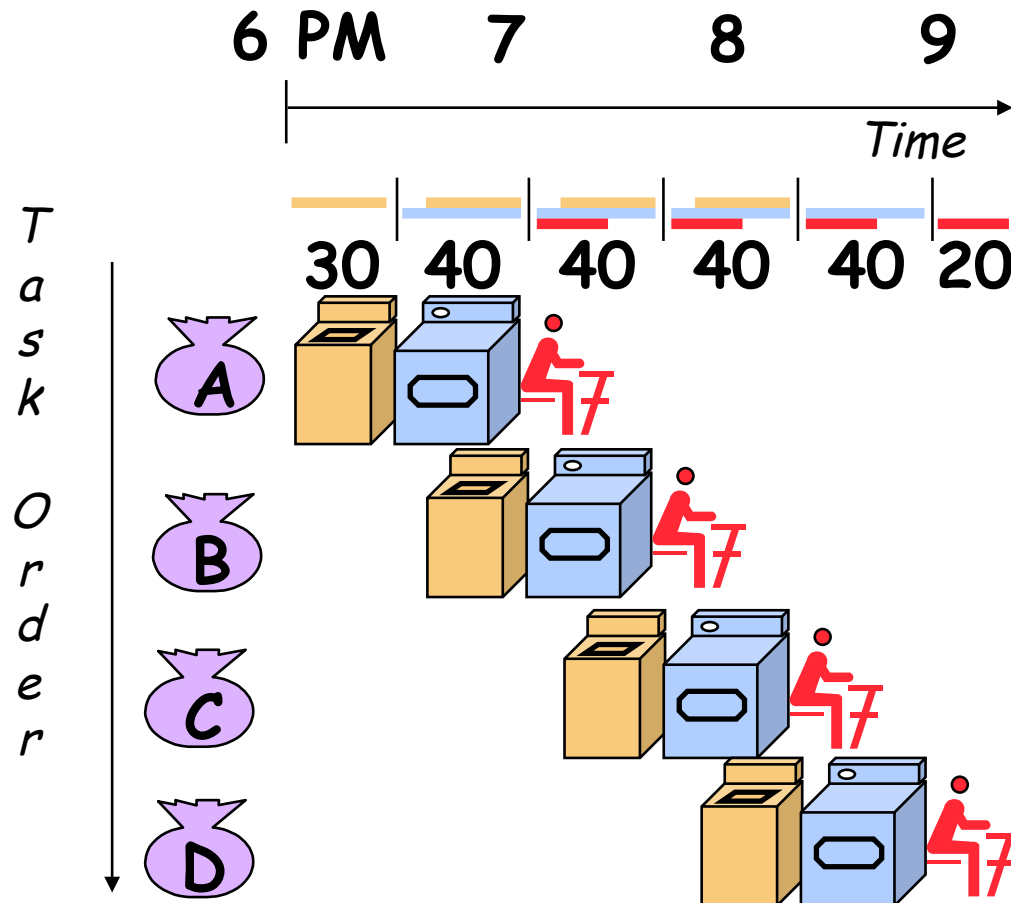
- Modern Processor Chips have \approx 1 billion transistors
 - Clearly must get them working in parallel
 - Question: how much of this parallelism must programmer understand?
- How do *uniprocessor* computer architectures extract parallelism?
 - By finding parallelism within instruction stream
 - Called "Instruction Level Parallelism" (ILP)
 - The theory: hide parallelism from programmer
- **Goal of Computer Architects until about 2002:**
 - **Hide Underlying Parallelism from everyone: OS, Compiler, Programmer**
- Examples of ILP techniques:
 - Pipelining: overlapping individual parts of instructions
 - Superscalar execution: do multiple things at same time
 - VLIW: Let compiler specify which operations can run in parallel
 - Vector Processing: Specify groups of similar (independent) operations
 - Out of Order Execution (OOO): Allow long operations to happen

What is Pipelining?

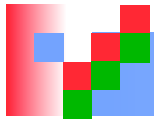


Dave Patterson's Laundry example: 4 people doing laundry

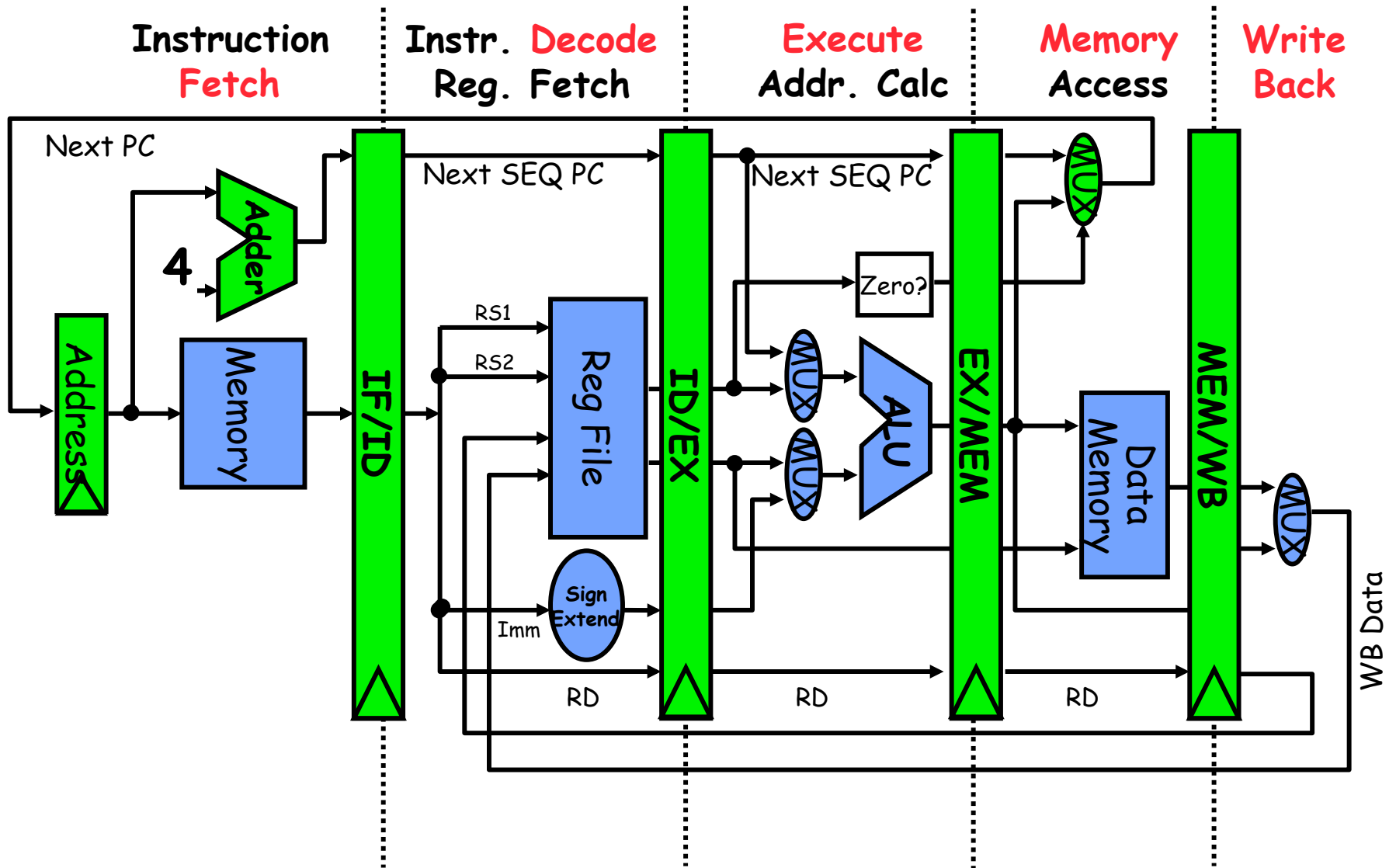
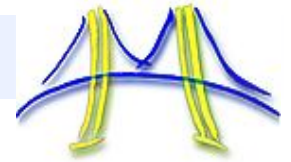
wash (30 min) + dry (40 min) + fold (20 min) = 90 min **Latency**

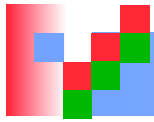


- In this example:
 - Sequential execution takes $4 * 90\text{min} = 6$ hours
 - Pipelined execution takes $30 + 4 * 40 + 20 = 3.5$ hours
- **Bandwidth** = loads/hour
 - $BW = 4/6$ l/h w/o pipelining
 - $BW = 4/3.5$ l/h w pipelining
 - $BW \leq 1.5$ l/h w pipelining, more total loads
- Pipelining helps **bandwidth** but not **latency** (90 min)
- Bandwidth limited by **slowest** pipeline stage
- Potential speedup = **Number of pipe stages**

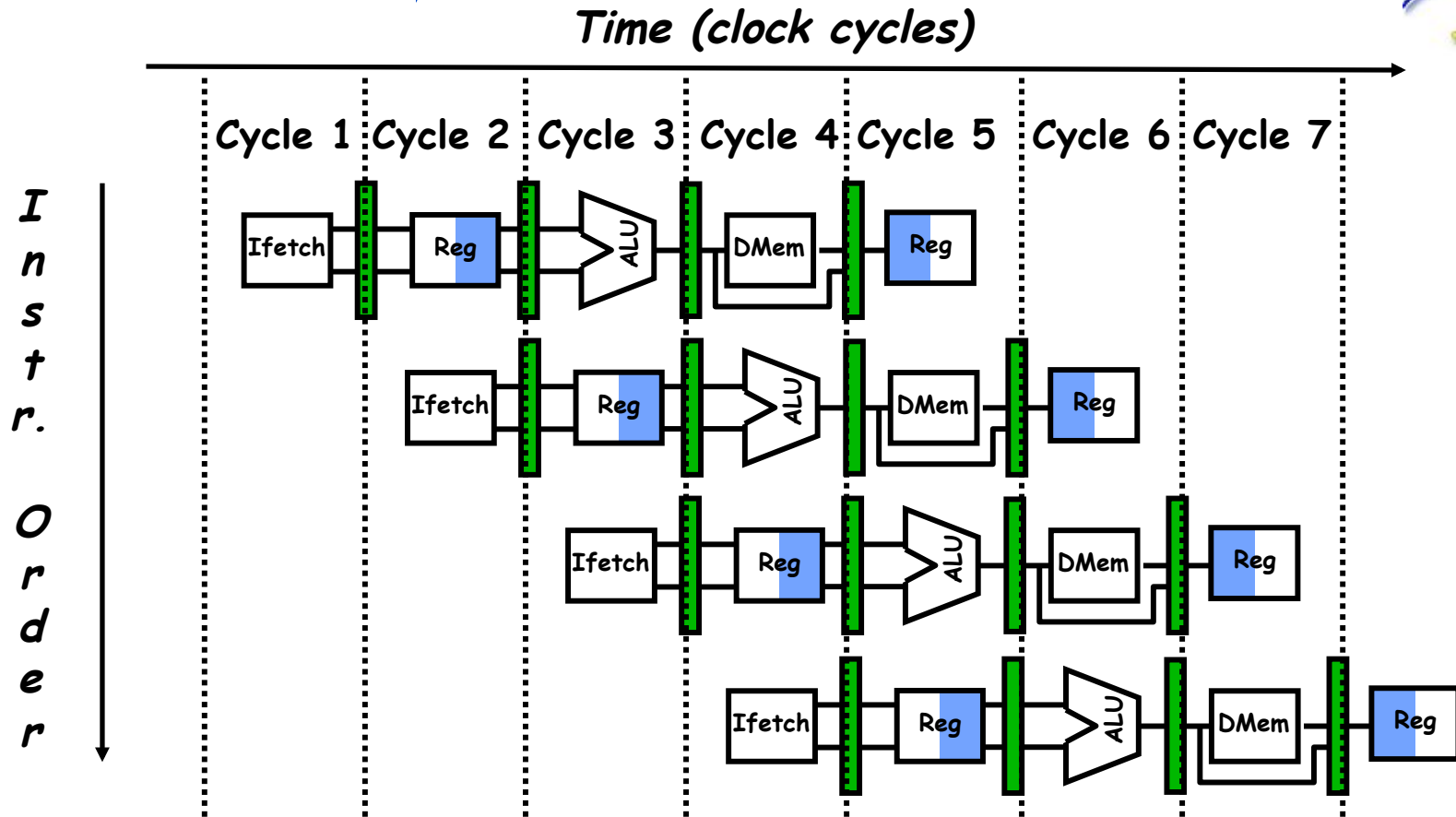
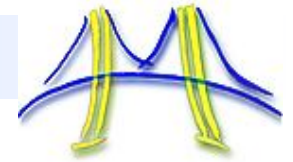


5 Steps of MIPS Pipeline

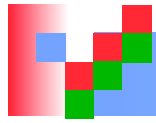




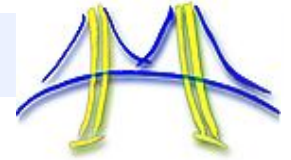
Visualizing The Pipeline



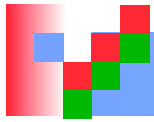
- In ideal case: $CPI \text{ (cycles/instruction)} = 1!$
 - On average, put one instruction into pipeline, get one out
- Superscalar: Launch more than one instruction/cycle
 - In ideal case, $CPI < 1$



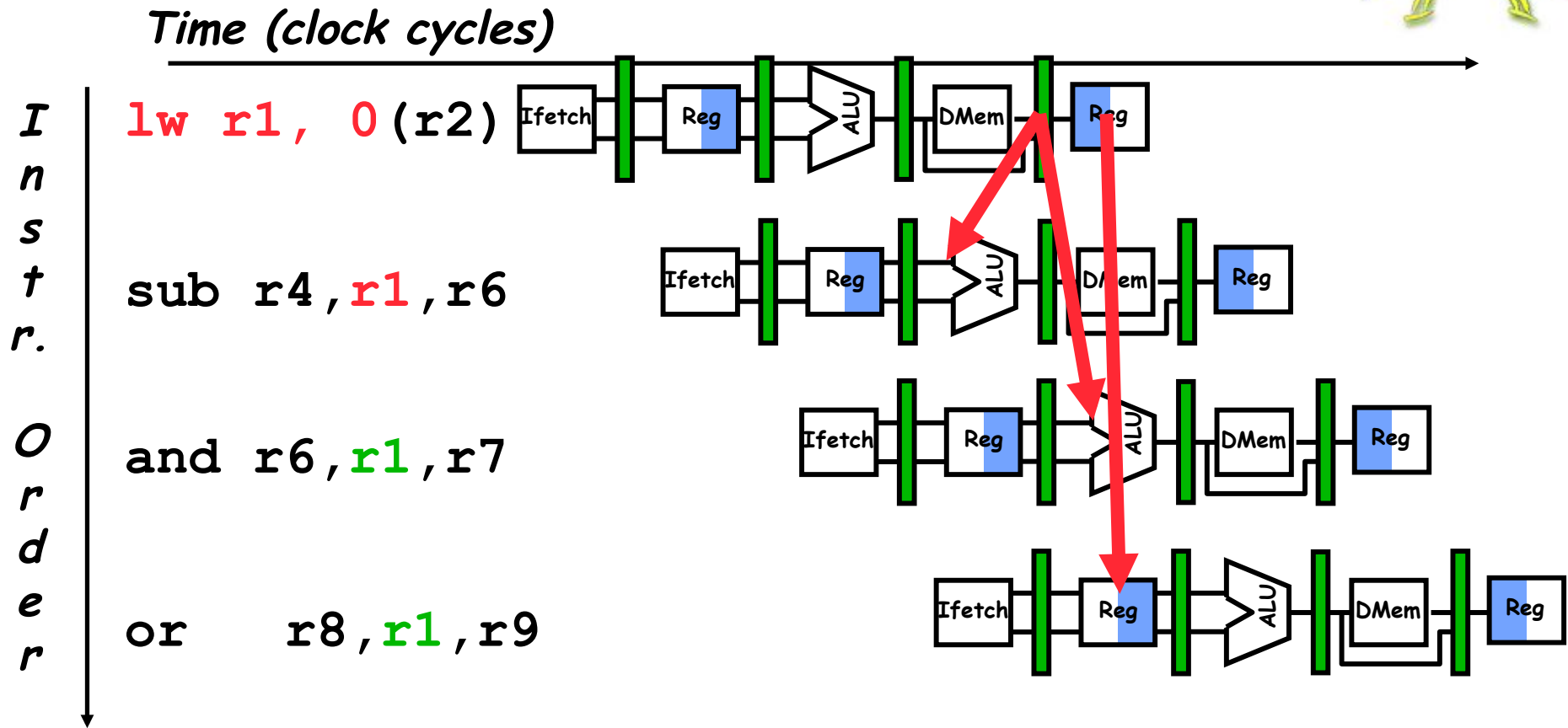
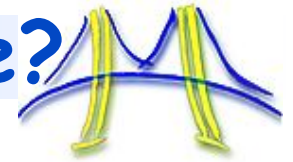
Limits to pipelining



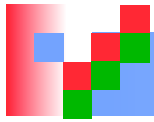
- Overhead prevents arbitrary division
 - Cost of latches (between stages) limits what can do within stage
 - Sets minimum amount of work/stage
- **Hazards** prevent next instruction from executing during its designated clock cycle
 - **Structural hazards:** attempt to use the same hardware to do two different things at once
 - **Data hazards:** Instruction depends on result of prior instruction still in the pipeline
 - **Control hazards:** Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
- Superscalar increases occurrence of hazards
 - More conflicting instructions/cycle



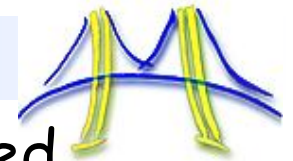
Data Hazard: Must go Back in Time?



- Data Dependencies between adjacent instructions
 - Must *wait* ("stall") for result to be done (No "back in time" exists!)
 - Net result is that $CPI > 1$
- Superscalar increases frequency of hazards



Out-of-Order (OOO) Execution



- Key idea: Allow instructions behind stall to proceed

```

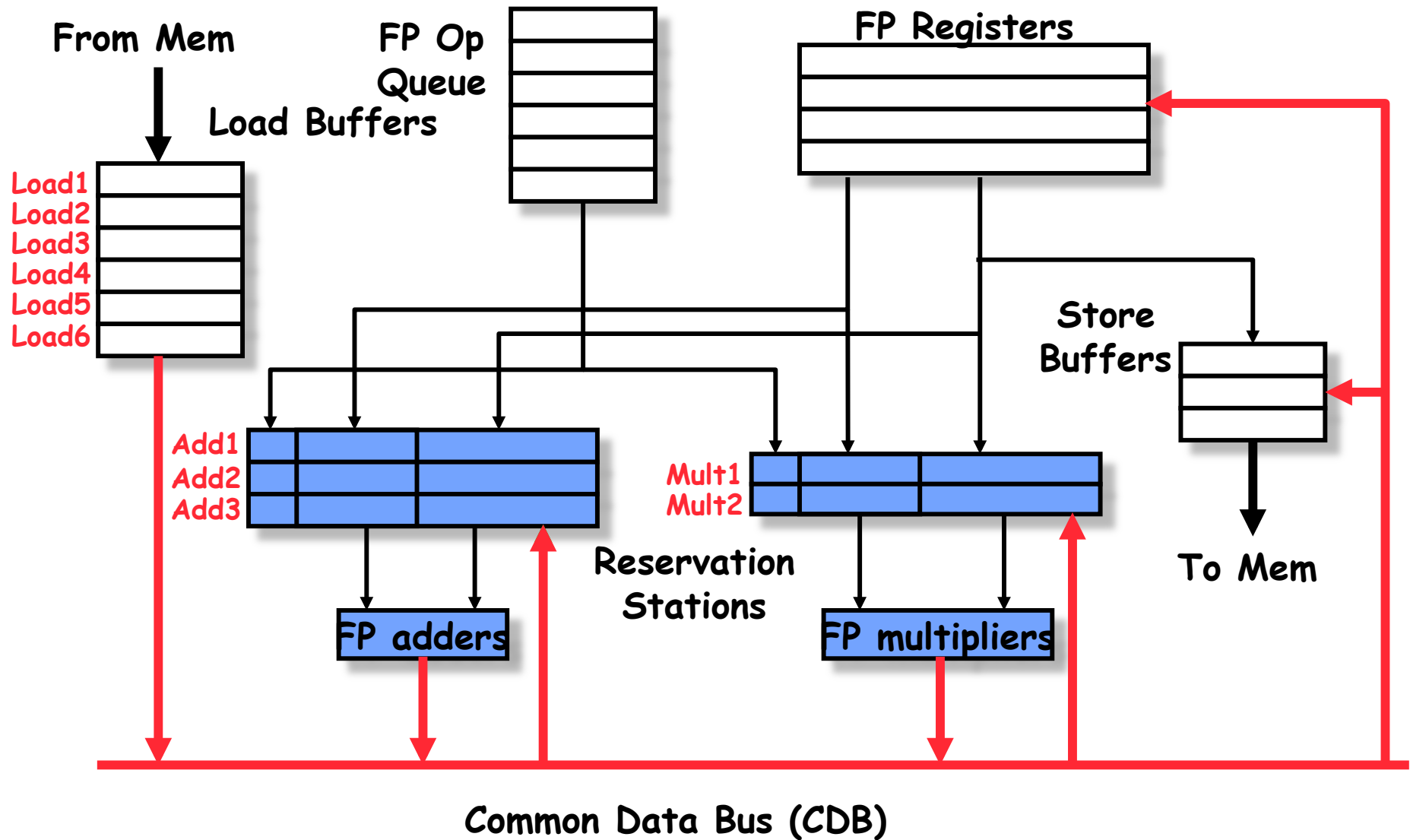
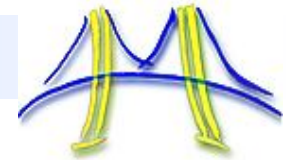
DIVD  F0, F2, F4
ADDD  F10, F0, F8
SUBD  F12, F8, F14

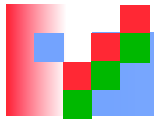
```

- Out-of-order execution \Rightarrow out-of-order completion.
- Dynamic Scheduling Issues from OOO scheduling:
 - Must match up results with consumers of instructions
 - Precise Interrupts

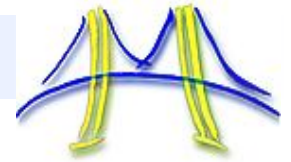
Instruction	Clock Cycle Number																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD F6,34(R2)	IF	ID	EX	MEM	WB												
LD F2,45(R3)		IF	ID	EX	MEM	WB											
MULTD F0,F2,F4			IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	MEM	WB
SUBD F8,F6,F2				IF	ID	A1	A2	MEM	WB								
DIVD F10,F0,F6					IF	ID	stall	stall	stall	stall	stall	stall	stall	stall	stall	D1	D2
ADDD F6,F8,F2						IF	ID	A1	A2	MEM	WB						

Basic Idea: Tomasulo Organization

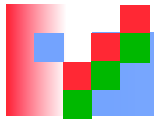




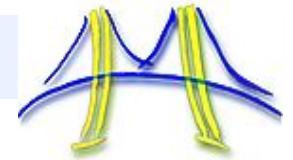
Modern ILP



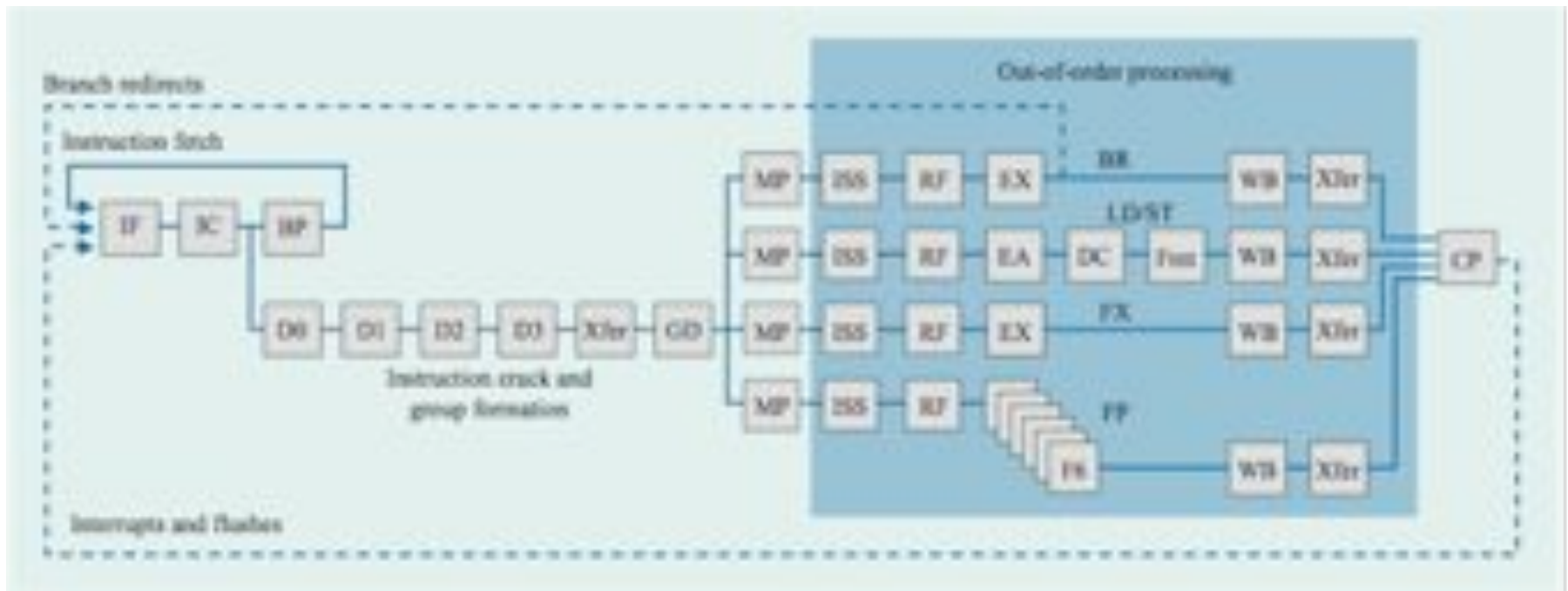
- Dynamically scheduled, out-of-order execution
 - Current microprocessors fetch 6-8 instructions per cycle
 - Pipelines are 10s of cycles deep \Rightarrow many overlapped instructions in execution at once, although work often discarded
- What happens:
 - Grab a bunch of instructions, determine all their dependences, eliminate dep's wherever possible, throw them all into the execution unit, let each one move forward as its dependences are resolved
 - Appears as if executed sequentially
- Dealing with Hazards: May need to *guess!*
 - Called "Speculative Execution"
 - Speculate on Branch results, Dependencies, even Values!
 - If correct, don't need to stall for result \Rightarrow yields performance
 - If not correct, waste time *and power*
 - Must be able to UNDO a result if guess is wrong
 - Problem: accuracy of guesses decreases with number of simultaneous instructions in pipeline
- Huge complexity
 - Complexity of many components scales as n^2 (issue width)
 - Power consumption big problem

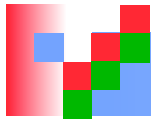


IBM Power 4

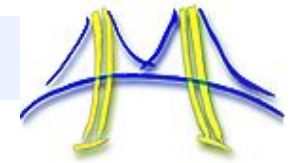


- **Combines: Superscalar and OOO**
- **Properties:**
 - 8 execution units in out-of-order engine, each may issue an instruction each cycle.
 - In-order Instruction Fetch, Decode (compute dependencies)
 - Reordering for in-order commit

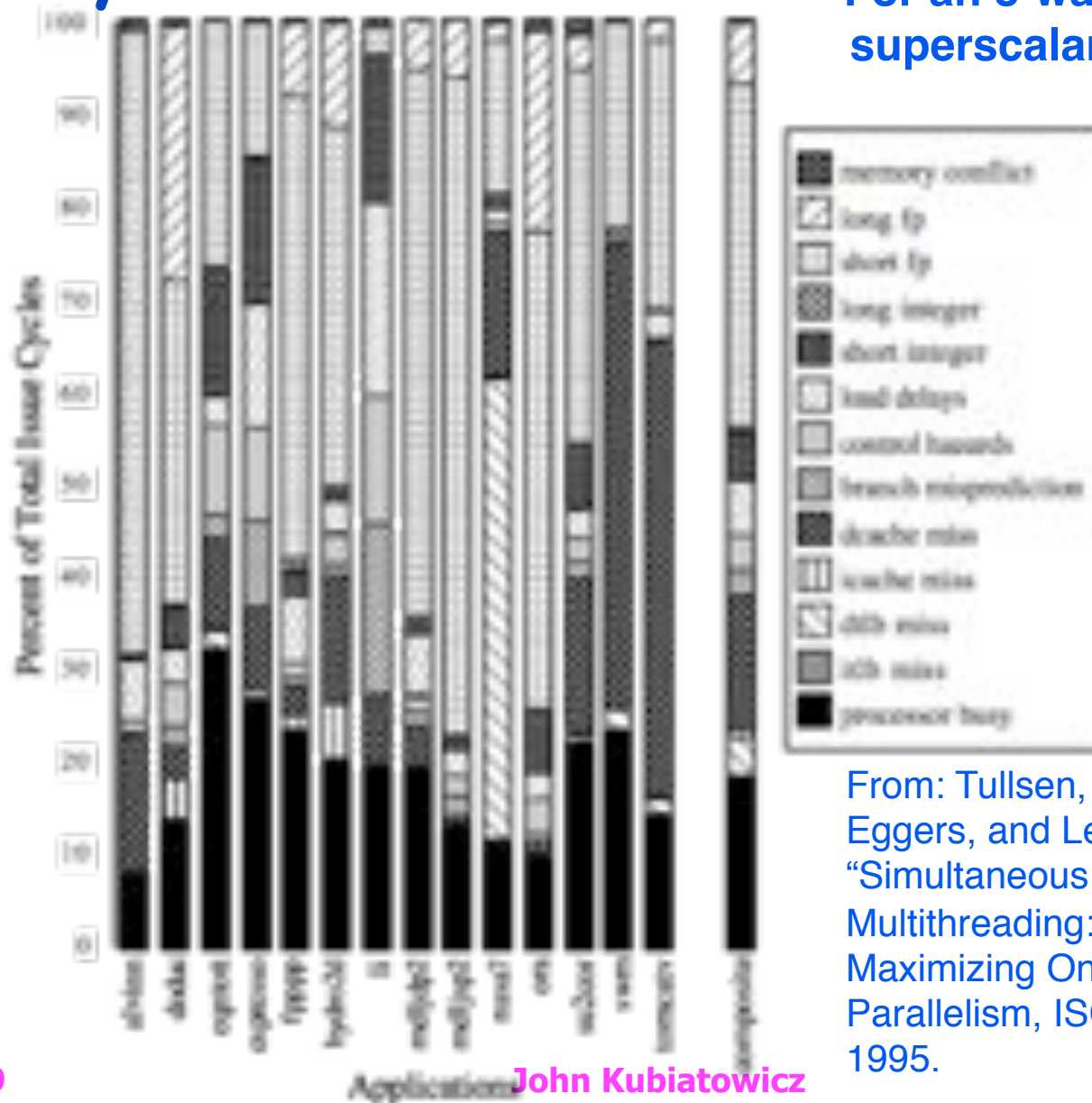




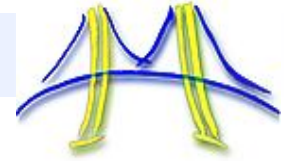
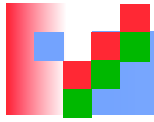
8-way OOO not Panacea: Many Resources IDLE!



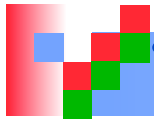
For an 8-way
superscalar.



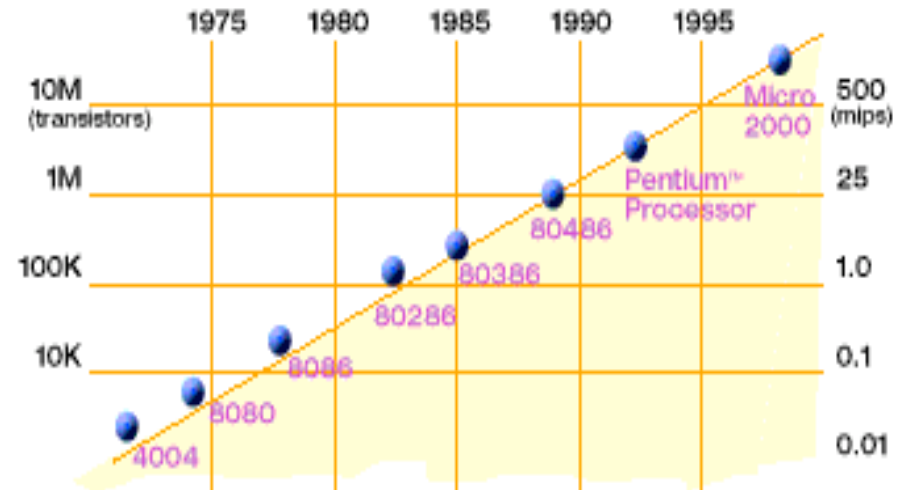
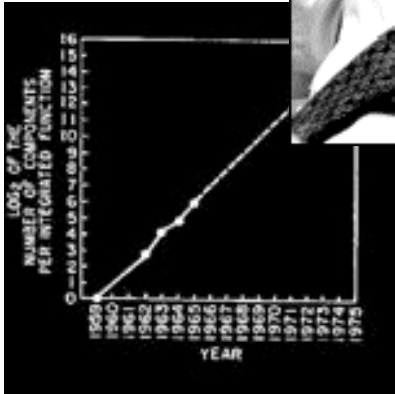
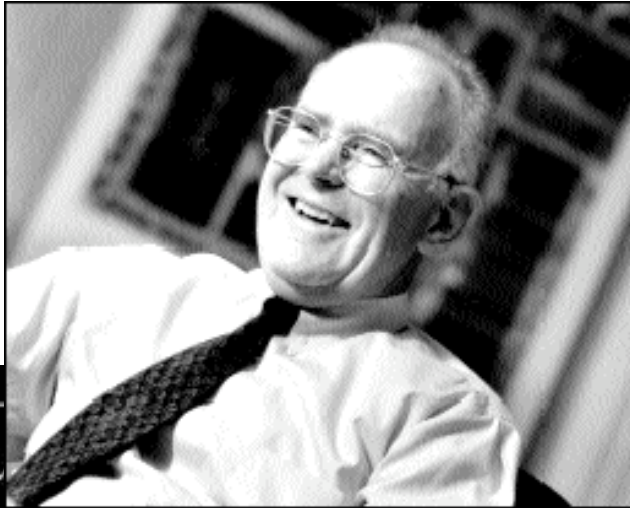
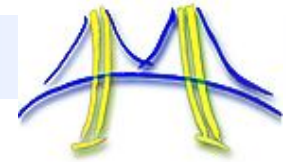
From: Tullsen,
Eggers, and Levy,
"Simultaneous
Multithreading:
Maximizing On-chip
Parallelism, ISCA
1995.

A large, multi-pointed blue starburst shape with a black outline, centered on the page. Inside the starburst, the text "Modern Limits" is written in a black, sans-serif font.

Modern Limits

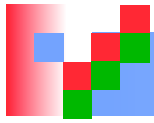


Technology Trends: Moore's Law

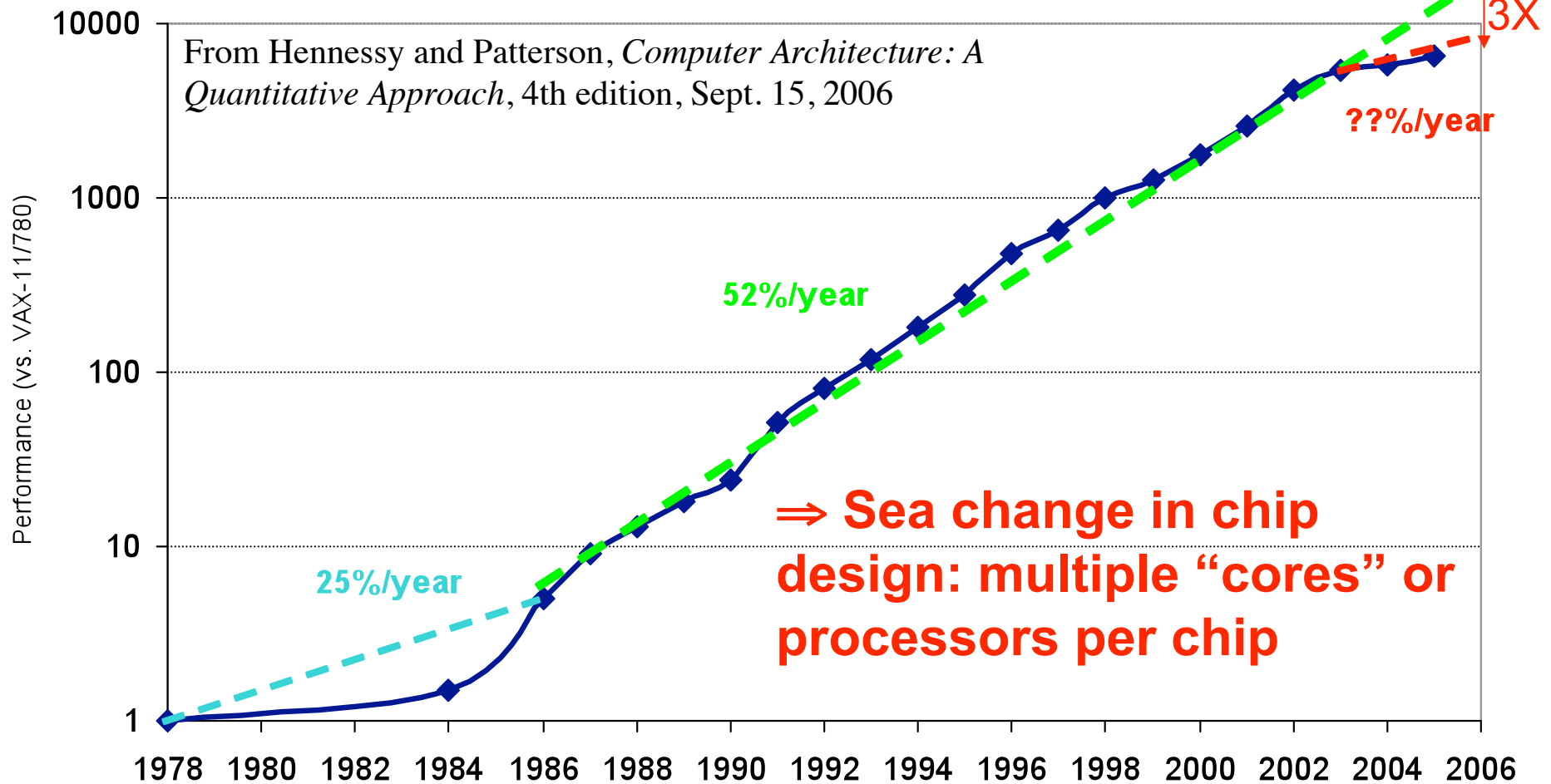
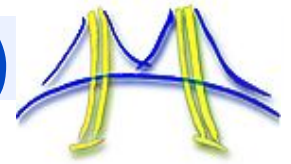


2X transistors/Chip Every 1.5 years
Called "**Moore's Law**"

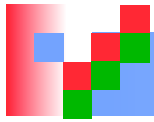
Microprocessors have become smaller, denser, and more powerful.



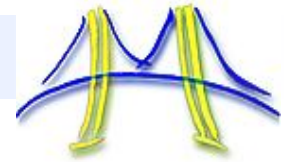
Uniprocessor Performance (SPECint)



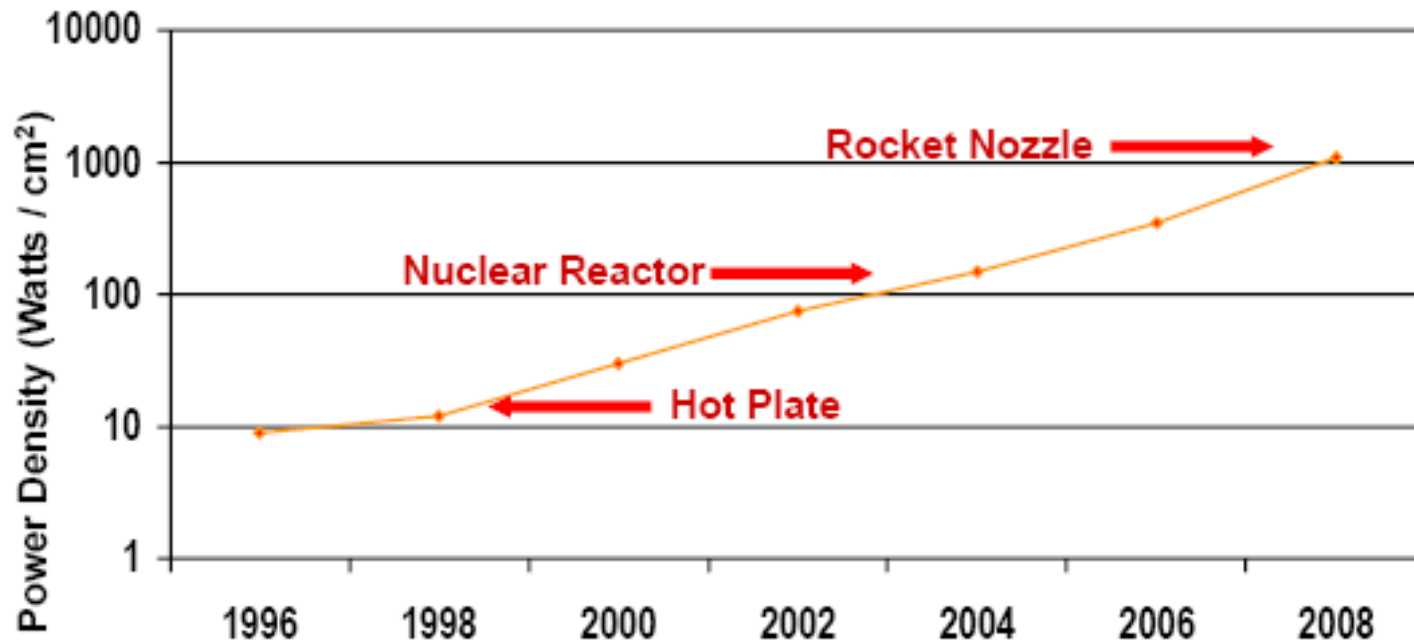
- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present



Limiting Force: Power Density

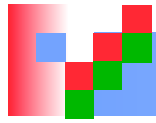


Moore's Law Extrapolation: Power Density for Leading Edge Microprocessors

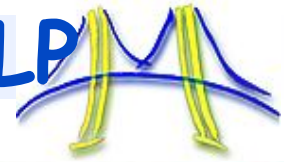


Power Density Becomes Too High to Cool Chips Inexpensively

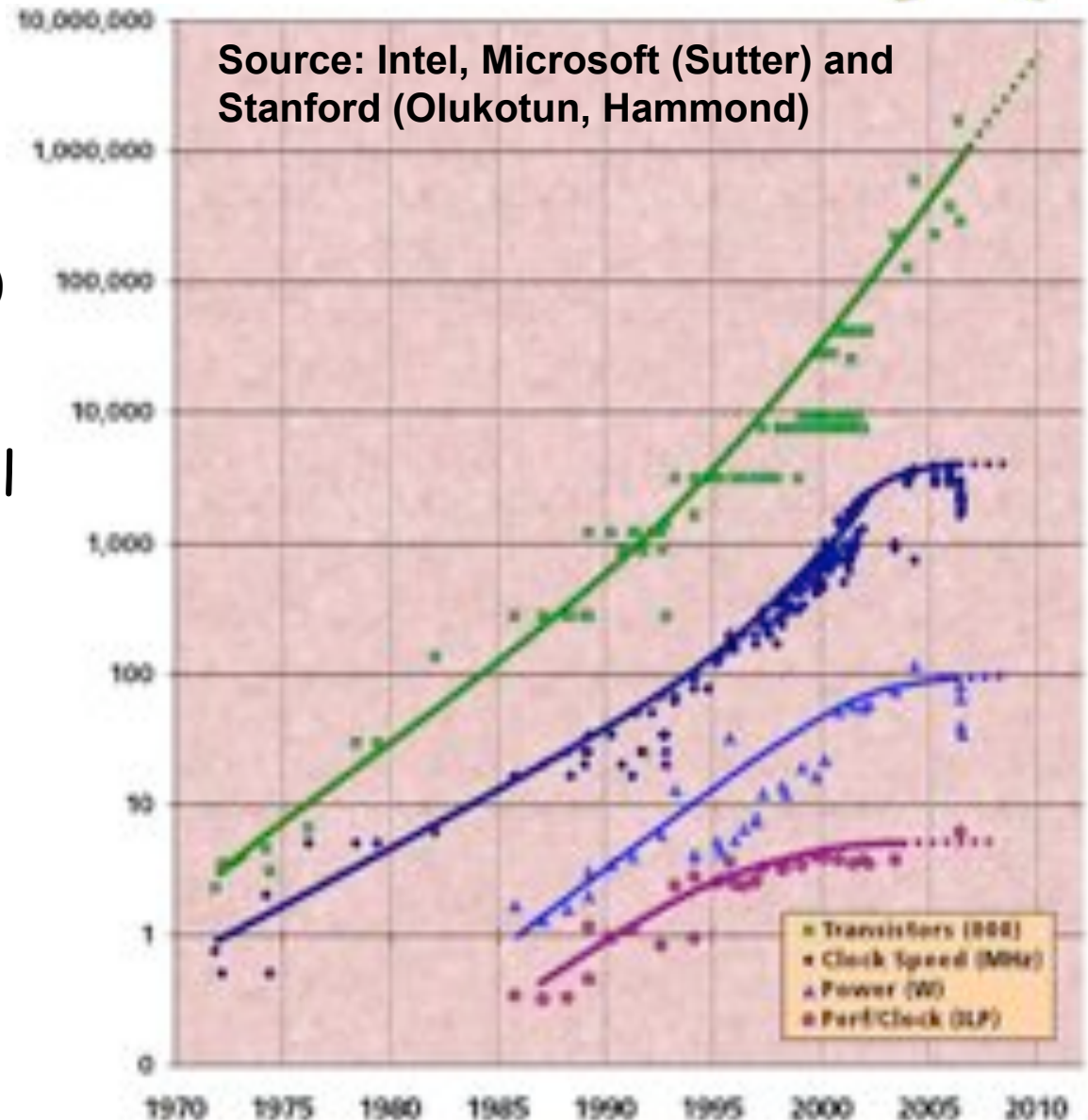
Source: Shekhar Borkar, Intel Corp

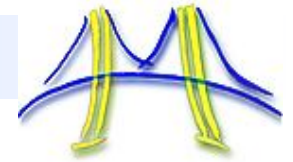
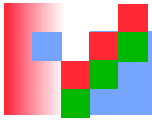


Limiting Forces: Clock Speed and ILP



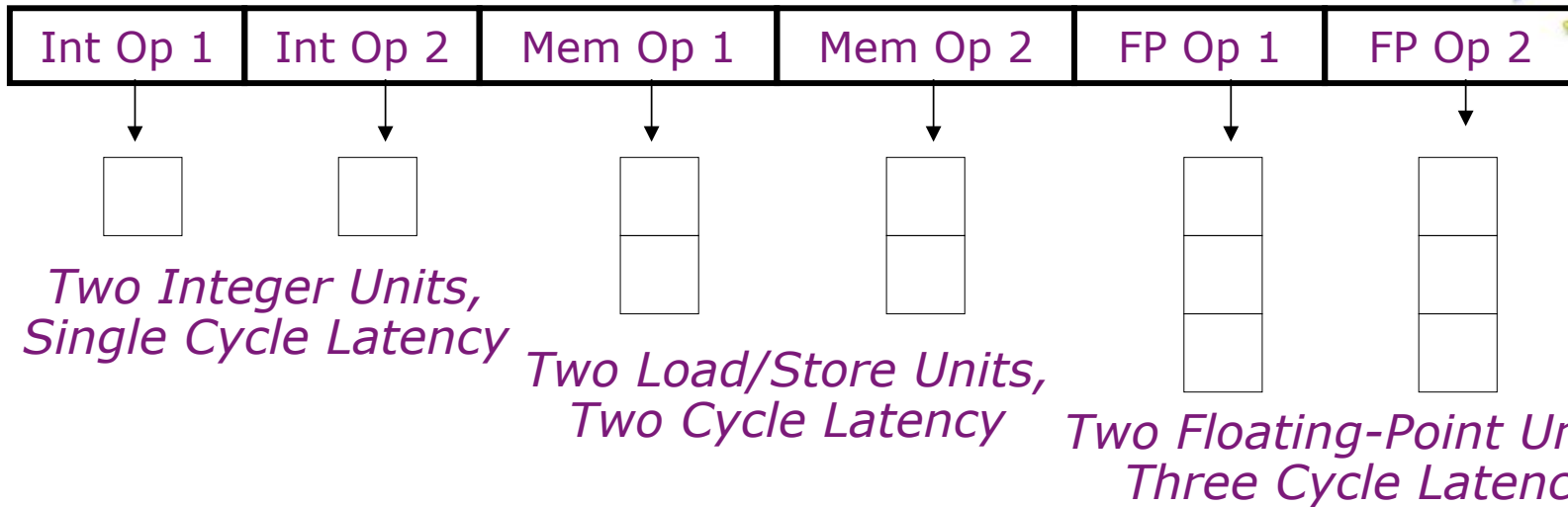
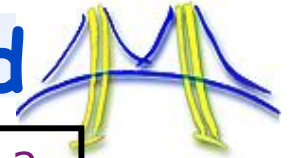
- Chip density is continuing increase
~2x every 2 years
 - Clock speed is not
 - # processors/chip (cores) may double instead
- There is little or no more Instruction Level Parallelism (ILP) to be found
 - Can no longer allow programmer to think in terms of a serial programming model
- Conclusion:
Parallelism must be exposed to software!



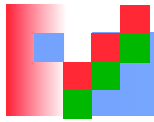


Relaxing the Sequential Model: VLIW

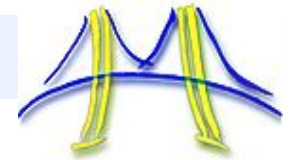
VLIW: Very Long Instruction Word



- Each "instruction" has explicit coding for multiple operations
 - In Itanium, grouping called a "packet"
 - In Transmeta, grouping called a "molecule" (with "atoms" as ops)
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires compiler guarantee of:
 - Parallelism within an instruction => no x-operation RAW check
 - No data use before data ready => no data interlocks
- Special compiler support must thus:
 - Extract parallelism
 - Prevent hazards from affecting results (through careful scheduling)
 - May require recompilation with each new version of hardware



Loop Unrolling in VLIW



```

Loop: LD    F0,0(R1)    ;F0=vector element
      ADDD  F4,F0,F2    ;add scalar from F2
      SD    0(R1),F4    ;store result
      SUBI  R1,R1,8     ;decrement pointer 8B (DW)
      BNEZ  R1,Loop    ;branch R1!=zero
      NOP                ;delayed branch slot

```

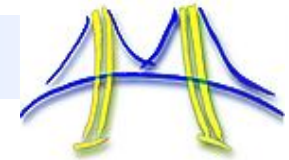
<i>Memory</i>	<i>Memory</i>	<i>FP</i>	<i>FP</i>	<i>Int. op/</i>
<i>reference 1</i>	<i>reference 2</i>	<i>operation 1</i>	<i>op. 2</i>	<i>branch</i>
LD F0,0(R1)	LD F6,-8(R1)			1
LD F10,-16(R1)	LD F14,-24(R1)			2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2	3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2	4
		ADDD F20,F18,F2	ADDD F24,F22,F2	5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2		6
SD -16(R1),F12	SD -24(R1),F16			7
SD -32(R1),F20	SD -40(R1),F24			8
SD -0(R1),F28				9
			SUBI R1,R1,#48	8
			BNEZ R1,LOOP	9

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

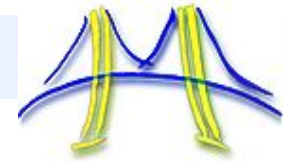
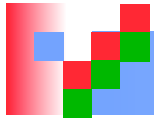
Software Pipelining with Loop Unrolling in VLIW



<i>Memory reference 1</i>	<i>Memory reference 2</i>	<i>FP operation 1</i>	<i>FP op. 2</i>	<i>Int. op/branch</i>	<i>Clock</i>
LD F0, -48(R1)	ST 0(R1), F4	ADDD F4, F0, F2			1
LD F6, -56(R1)	ST -8(R1), F8	ADDD F8, F6, F2		SUBI R1, R1, #24	2
LD F10, -40(R1)	ST 8(R1), F12	ADDD F12, F10, F2		BNEZ R1, LOOP	3

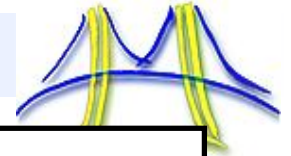
- Software pipelined across 9 iterations of original loop
 - In each iteration of above loop, we:
 - » Store to m, m-8, m-16 (iterations I-3, I-2, I-1)
 - » Compute for m-24, m-32, m-40 (iterations I, I+1, I+2)
 - » Load from m-48, m-56, m-64 (iterations I+3, I+4, I+5)
- 9 results in 9 cycles, or 1 clock per iteration
- Average: 3.3 ops per clock, 66% efficiency

Note: Need less registers for software pipelining (only using 7 registers here, was using 15)



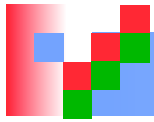
Relaxing the Sequential Model: Vectors/SIMD

Vector Code Example

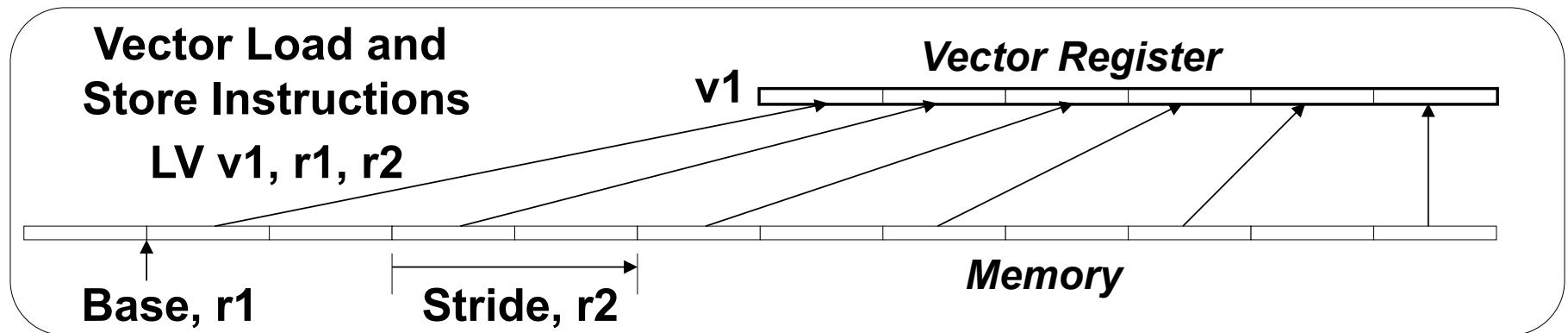
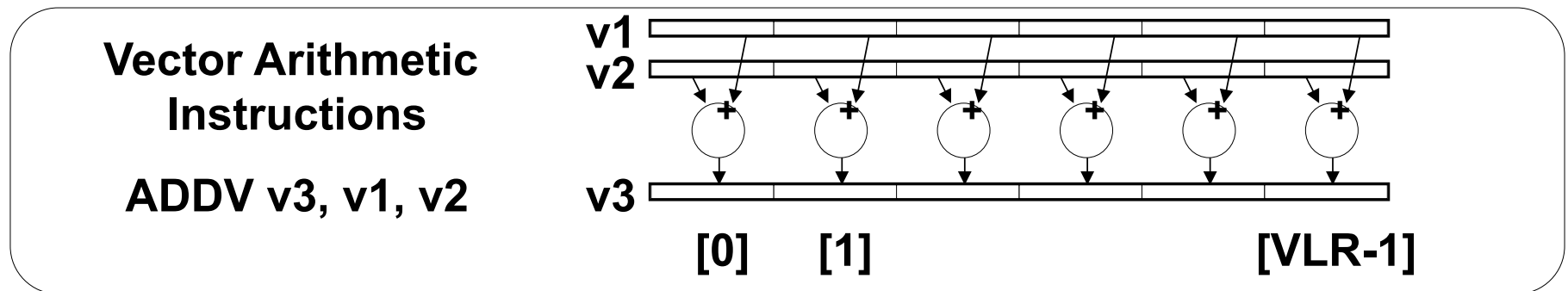
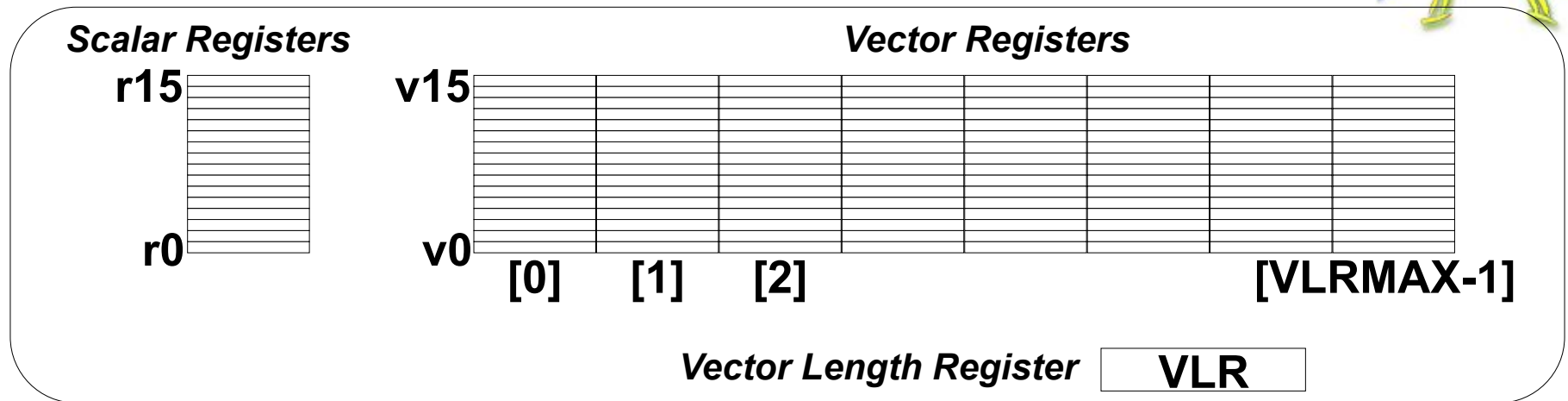
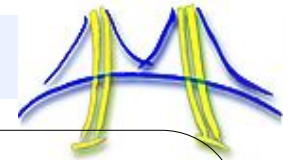


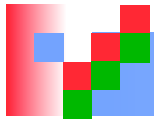
# C code	# Scalar Code	# Vector Code
<pre>for (i=0; i<64; i++) C[i] = A[i] + B[i];</pre>	<pre>LI R4, 64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop</pre>	<pre>LI VLR, 64 LV V1, R1 LV V2, R2 ADDV.D V3, V1, V2 SV V3, R3</pre>

- Require programmer (or compiler) to identify parallelism
 - Hardware does not need to re-extract parallelism
- Many multimedia/HPC applications are natural consumers of vector processing

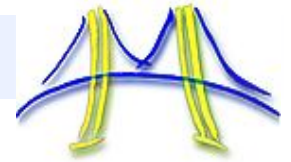


Vector Programming Model

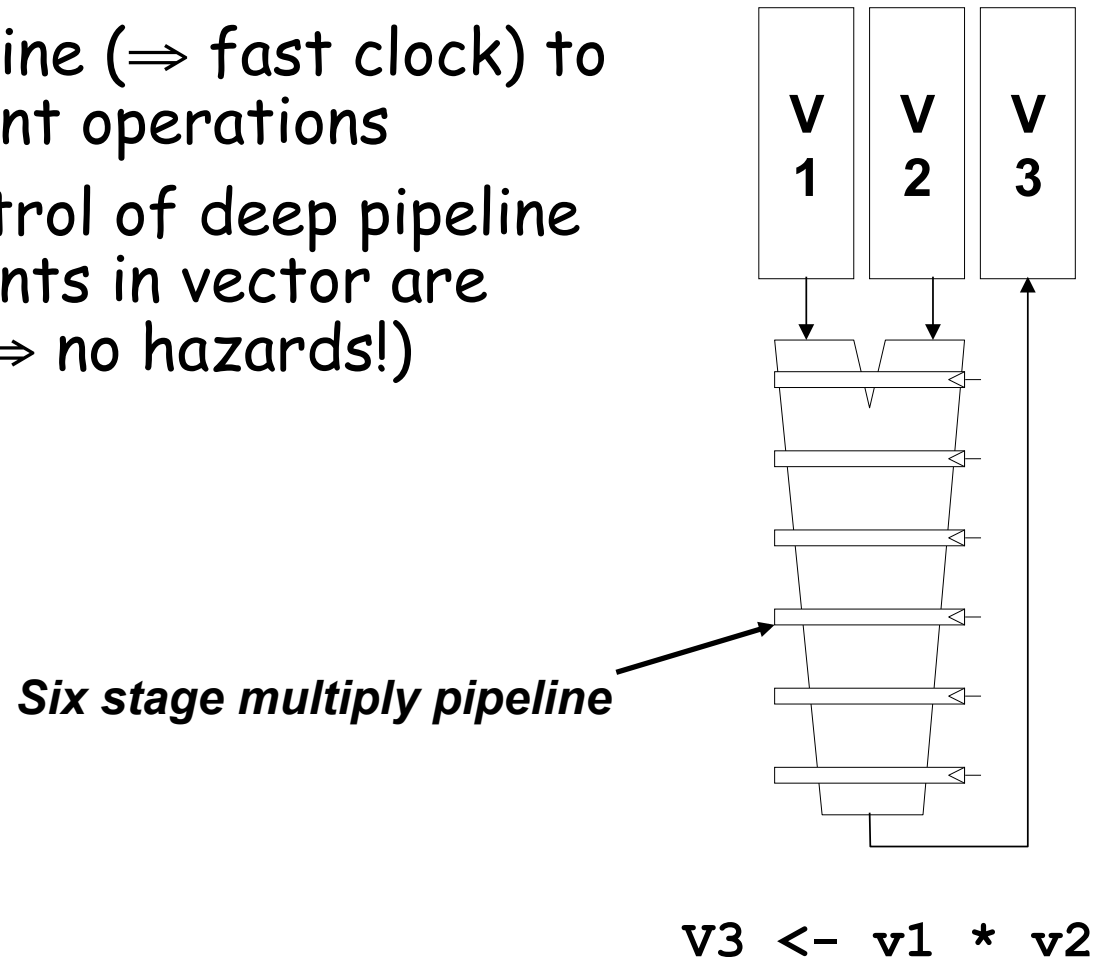


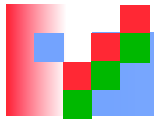


Vector Arithmetic Execution

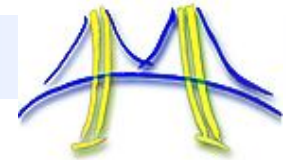


- Use deep pipeline (\Rightarrow fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (\Rightarrow no hazards!)



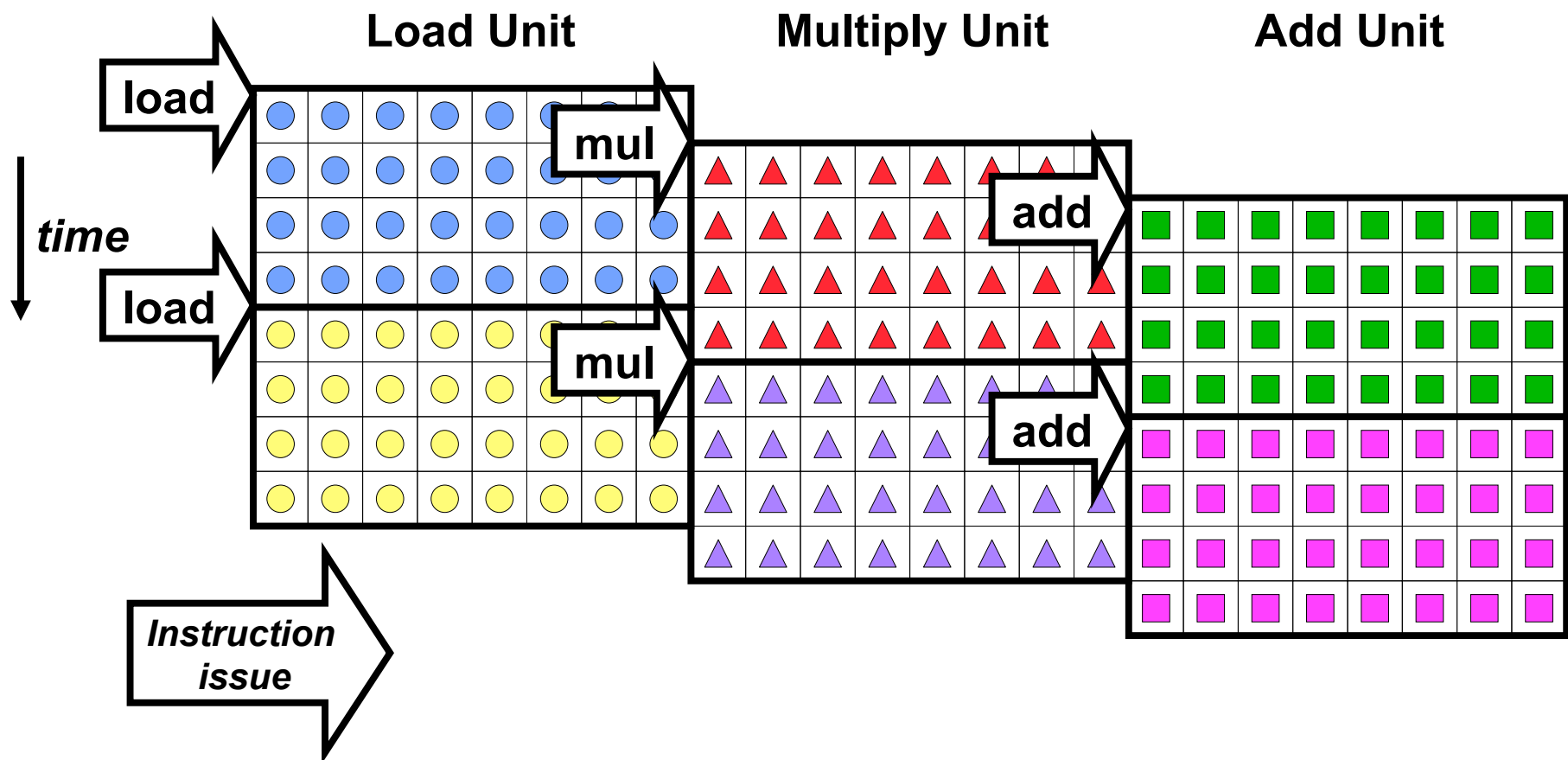


Vector Instruction Parallelism



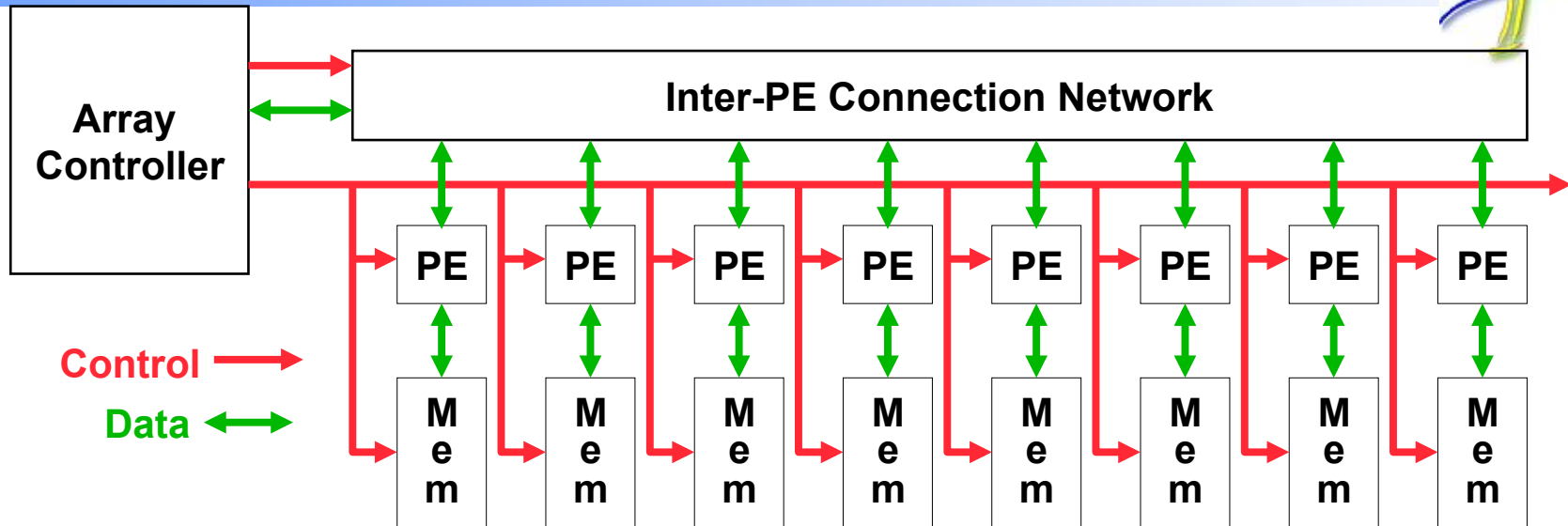
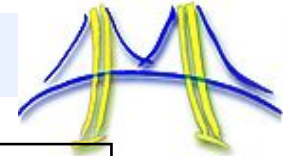
Can overlap execution of multiple vector instructions

- Consider machine with 32 elements per vector register and 8 lanes:

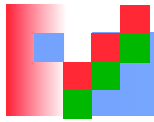


Complete 24 operations/cycle while issuing 1 short instruction/cycle

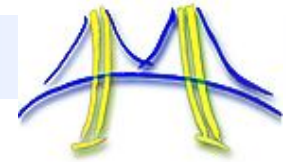
SIMD Architecture



- Single Instruction Multiple Data (SIMD)
- Central controller broadcasts instructions to multiple processing elements (PEs)
 - Only requires one controller for whole array
 - Only requires storage for one copy of program
 - All computations fully synchronized
- Recent Return to Popularity:
 - GPU (Graphics Processing Units) have SIMD properties
 - However, also multicore behavior, so mix of SIMD and MIMD (more later)
- Dual between Vector and SIMD execution

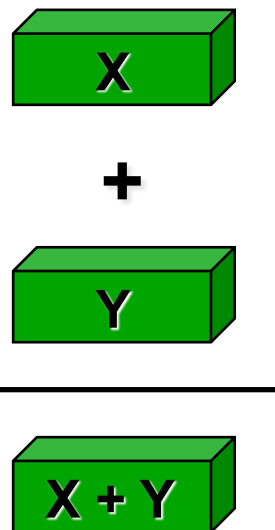


Pseudo SIMD: (Poor-Man's SIMD?)



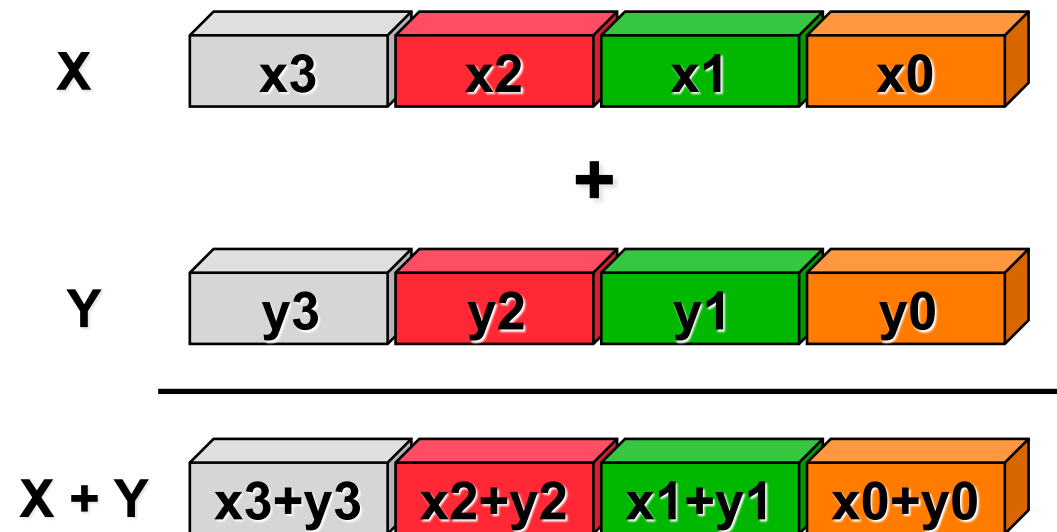
- Scalar processing

- traditional mode
- one operation produces one result

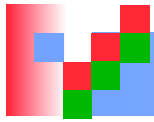


- SIMD processing (Intel)

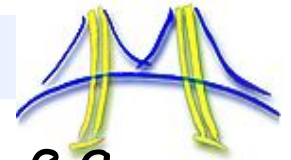
- with SSE / SSE2
- one operation produces multiple results



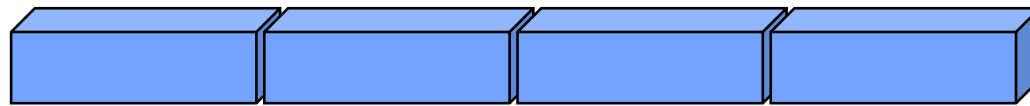
Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation



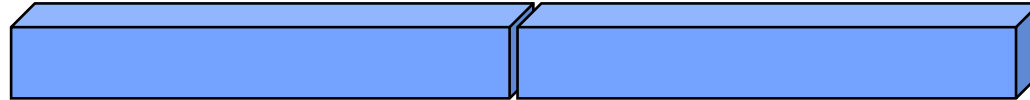
E.g.: SSE / SSE2 SIMD on Intel



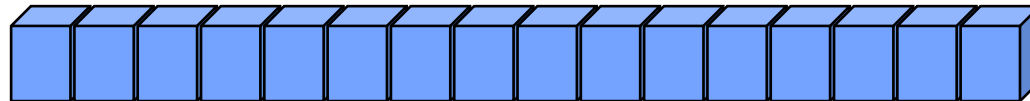
- SSE2 data types: anything that fits into 16 bytes, e.g.,



4x floats

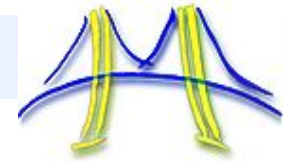
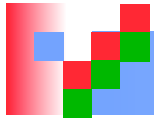


2x doubles

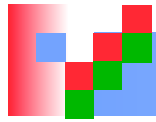


16x bytes

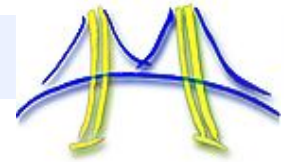
- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
 - Need to be contiguous in memory and aligned
 - Some instructions to move data from one part of register to another
- In theory, the compiler understands all of this
 - When compiling, it will rearrange instructions to get a good "schedule" that maximizes pipelining, uses FMAs and SIMD
 - It works with the mix of instructions inside an inner loop or other block of code
- But in practice the compiler may need your help



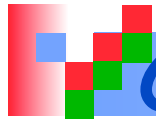
Relaxing the Sequential Model: Multithreading



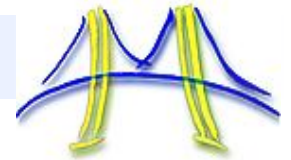
Thread Level Parallelism (TLP)



- ILP exploits implicit parallel operations within a loop or straight-line code segment
- TLP explicitly represented by the use of multiple threads of execution that are inherently parallel
 - Threads can be on a single processor
 - Or, on multiple processors
- Goal: Use multiple instruction streams to improve
 1. Throughput of computers that run many programs
 2. Execution time of multi-threaded programs



Common Notions of Thread Creation



- **cobegin/coend**

```
cobegin
  job1 (a1) ;
  job2 (a2) ;
coend
```

- Statements in block may run in parallel
- cobegins may be nested
- Scoped, so you cannot have a missing coend

- **fork/join**

```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```

- Forked procedure runs in parallel
- Wait at join point if it's not finished

- **future**

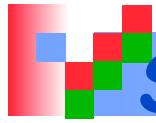
```
v = future(job1(a1));
... = ...v...;
```

- Future expression evaluated in parallel
- Attempt to use return value will wait

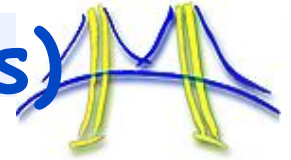
- **Cobegin cleaner than fork, but fork is more general**

- **Threads expressed in the code may not turn into independent computations**

- Only create threads if processors idle
- Example: Thread-stealing runtimes such as cilk

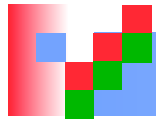


Simple Threading Example (pThreads)

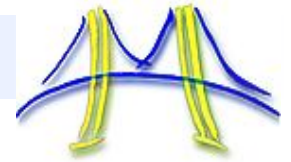


```
void* SayHello(void *foo) {
    printf( "Hello, world!\n" );
    return NULL;
}

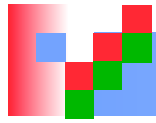
int main() {
    pthread_t threads[16];
    int tn;
    for(tn=0; tn<16; tn++) {
        pthread_create(&threads[tn], NULL, SayHello, NULL);
    }
    for(tn=0; tn<16 ; tn++) {
        pthread_join(threads[tn], NULL);
    }
    return 0;
}
```



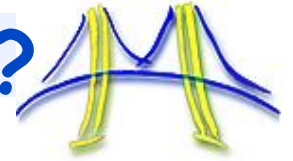
Multithreaded Execution



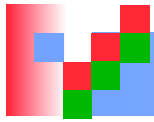
- **Multitasking operating system:**
 - Gives "illusion" that multiple things happening at same time
 - Switches at a coarse-grained time quanta (for instance: 10ms)
- **Hardware Multithreading: multiple threads share processor simultaneously (with little OS help)**
 - Hardware does switching
 - » HW for fast thread switch in small number of cycles
 - » much faster than OS switch which is 100s to 1000s of clocks
 - Processor duplicates independent state of each thread
 - » e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
 - Memory shared through the virtual memory mechanisms, which already support multiple processes
- **When to switch between threads?**
 - Alternate instruction per thread (fine grain)
 - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)



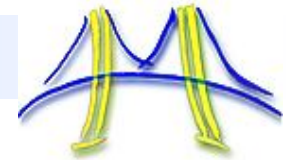
What about combining ILP and TLP?



- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP benefit from exploiting TLP?
 - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
 - TLP used as a source of independent instructions that might keep the processor busy during stalls
 - TLP be used to occupy functional units that would otherwise lie idle when insufficient ILP exists
- Called "Simultaneous Multithreading"
 - Intel renamed this "Hyperthreading"



Simultaneous Multi-threading ...



One thread, 8 units

Cycle M M FX FX FP FP BR CC

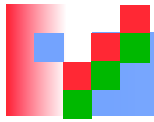
1	█							█
2	█	█					█	
3			█	█				
4								
5								
6								
7	█		█		█			
8		█		█				
9			█					

Two threads, 8 units

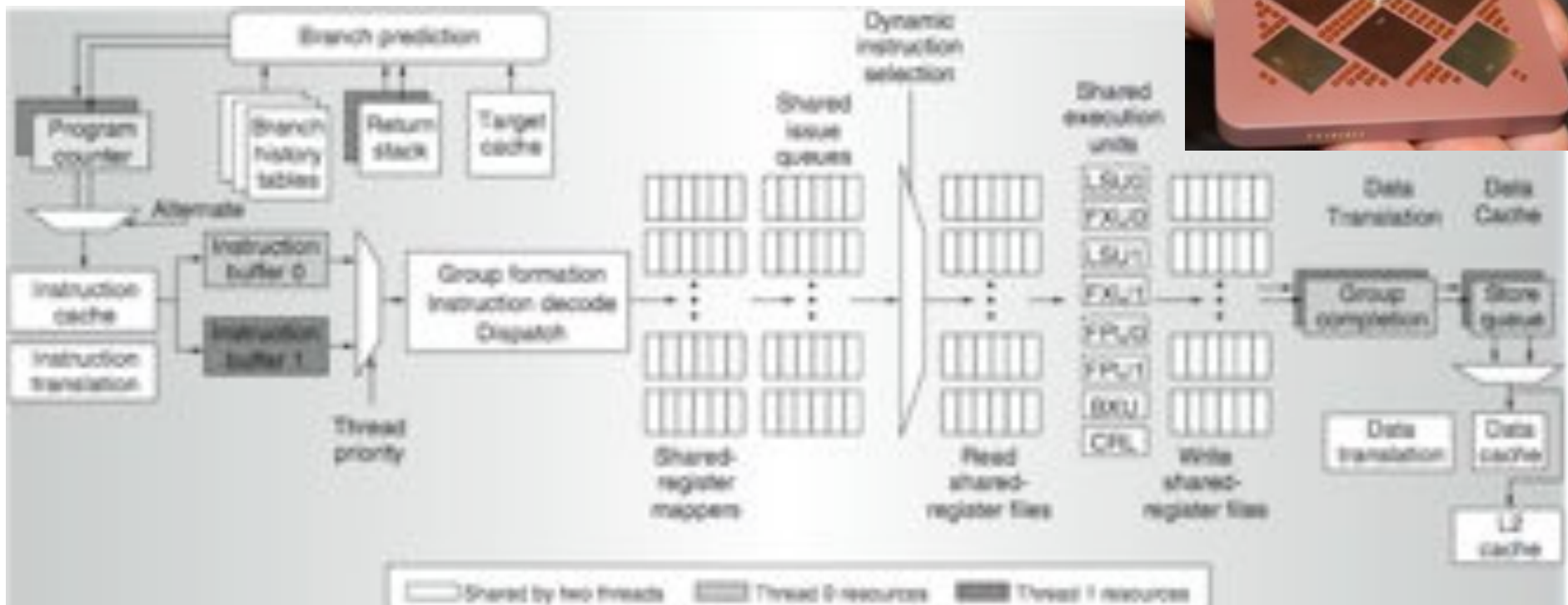
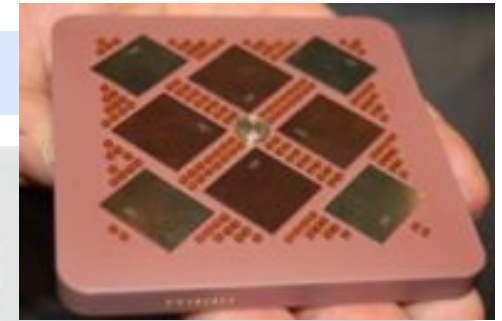
Cycle M M FX FX FP FP BR CC

1	█	█	█					█
2	█	█	█			█	█	
3	█			█	█			
4	█	█				█		
5		█						█
6								
7	█		█	█	█	█		
8		█		█	█	█		
9	█	█		█		█		

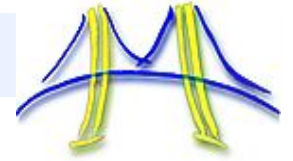
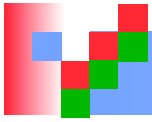
M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes



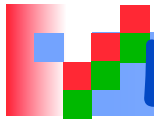
Power 5 dataflow ...



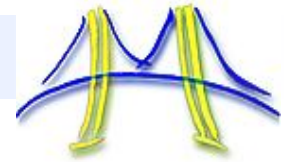
- Why only two threads?
 - With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck
- Cost:
 - The Power5 core is about 24% larger than the Power4 core because of the addition of SMT support



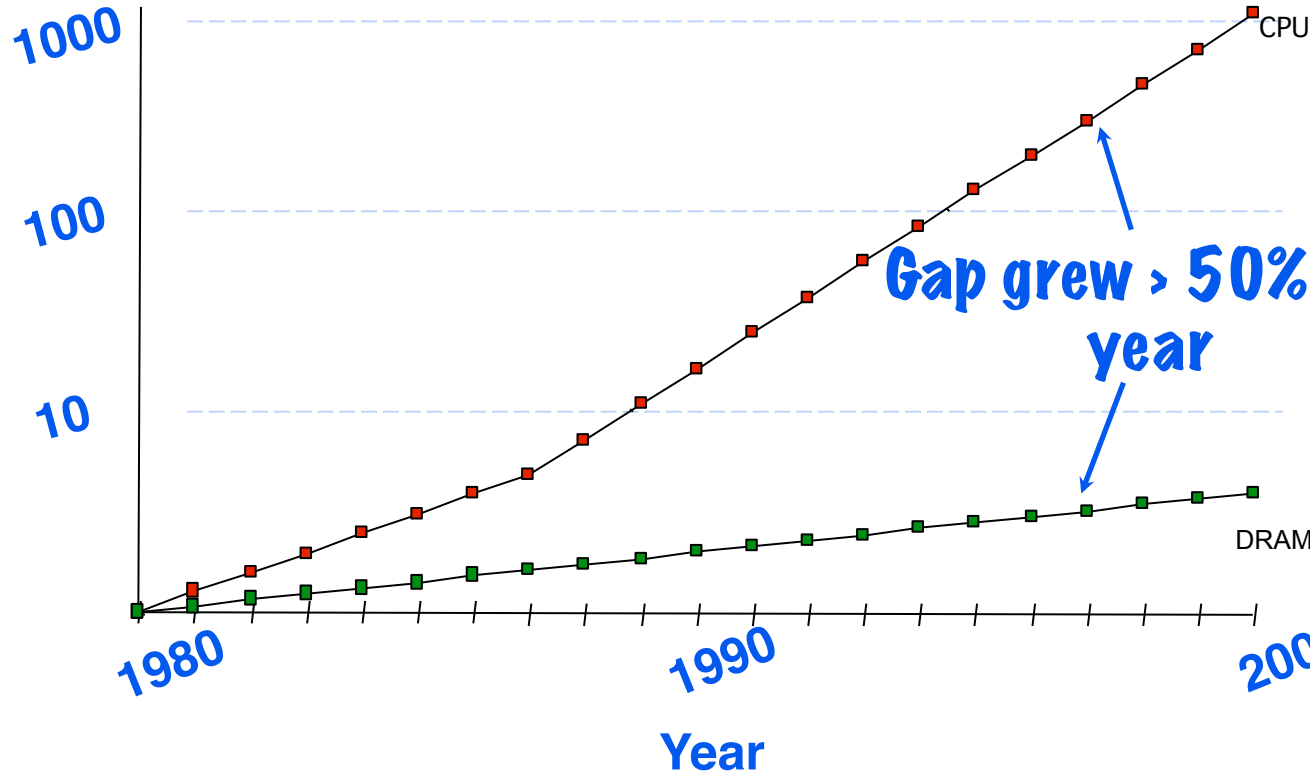
The Sequential Memory System



Limiting Force: Memory System



Performance
(1/latency)



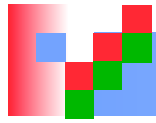
CPU

60% per yr
2X in 1.5 yrs

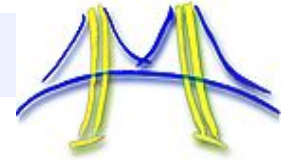
DRAM

5.5-7% per yr
<2X in 10 yrs

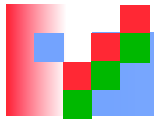
- How do architects address this gap?
 - Put small, fast "cache" memories between CPU and DRAM.
 - Create a "memory hierarchy"



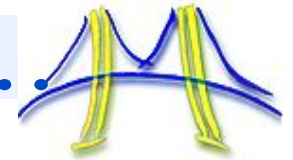
The Principle of Locality



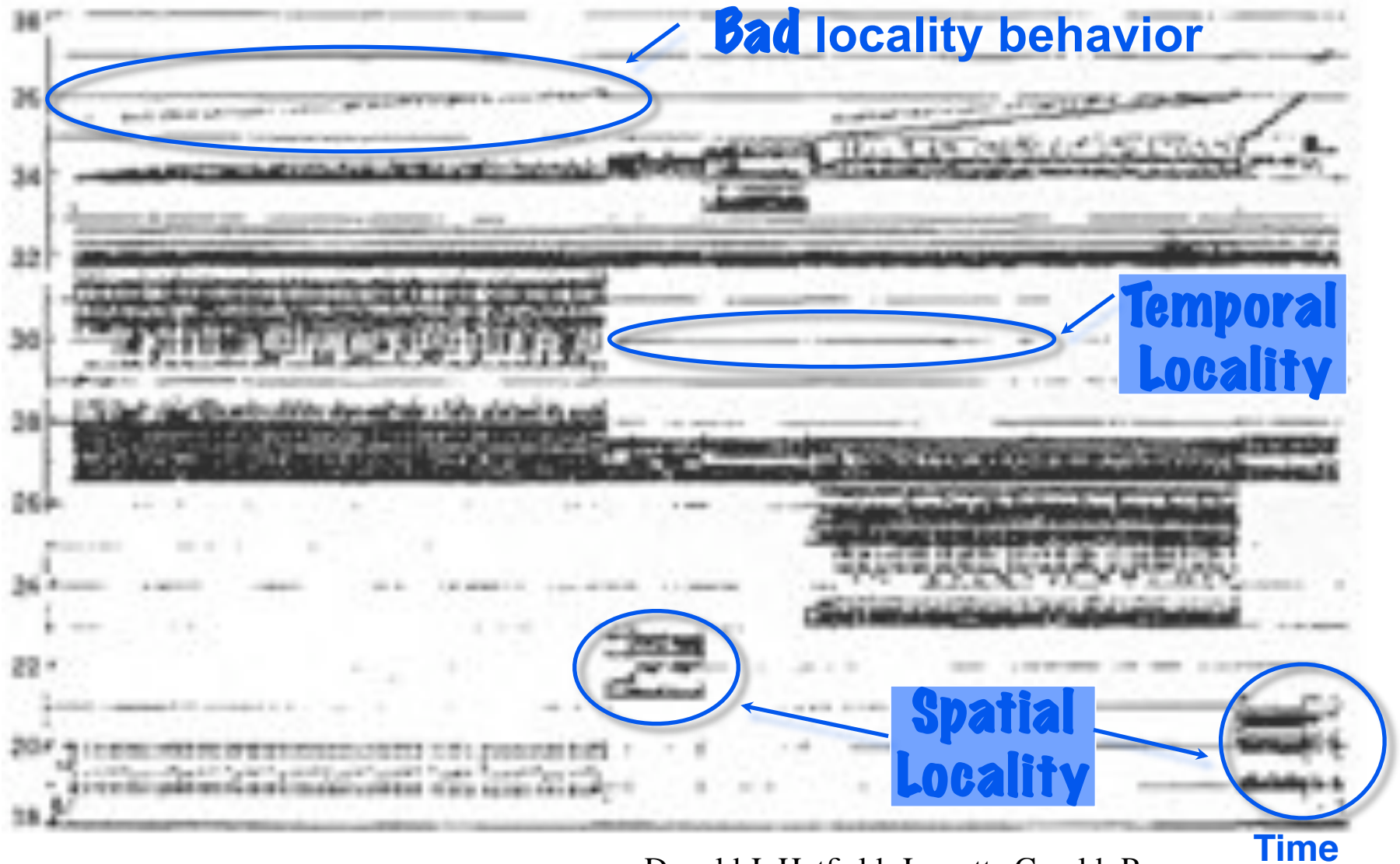
- The Principle of Locality:
 - Program access a relatively small portion of the address space at any instant of time
- Two Different Types of Locality:
 - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
 - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- Last 25 years, HW relied on locality for speed



Programs with locality cache well ..

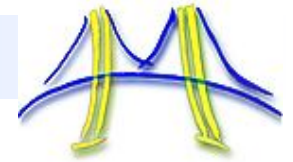


Memory Address (one dot per access)

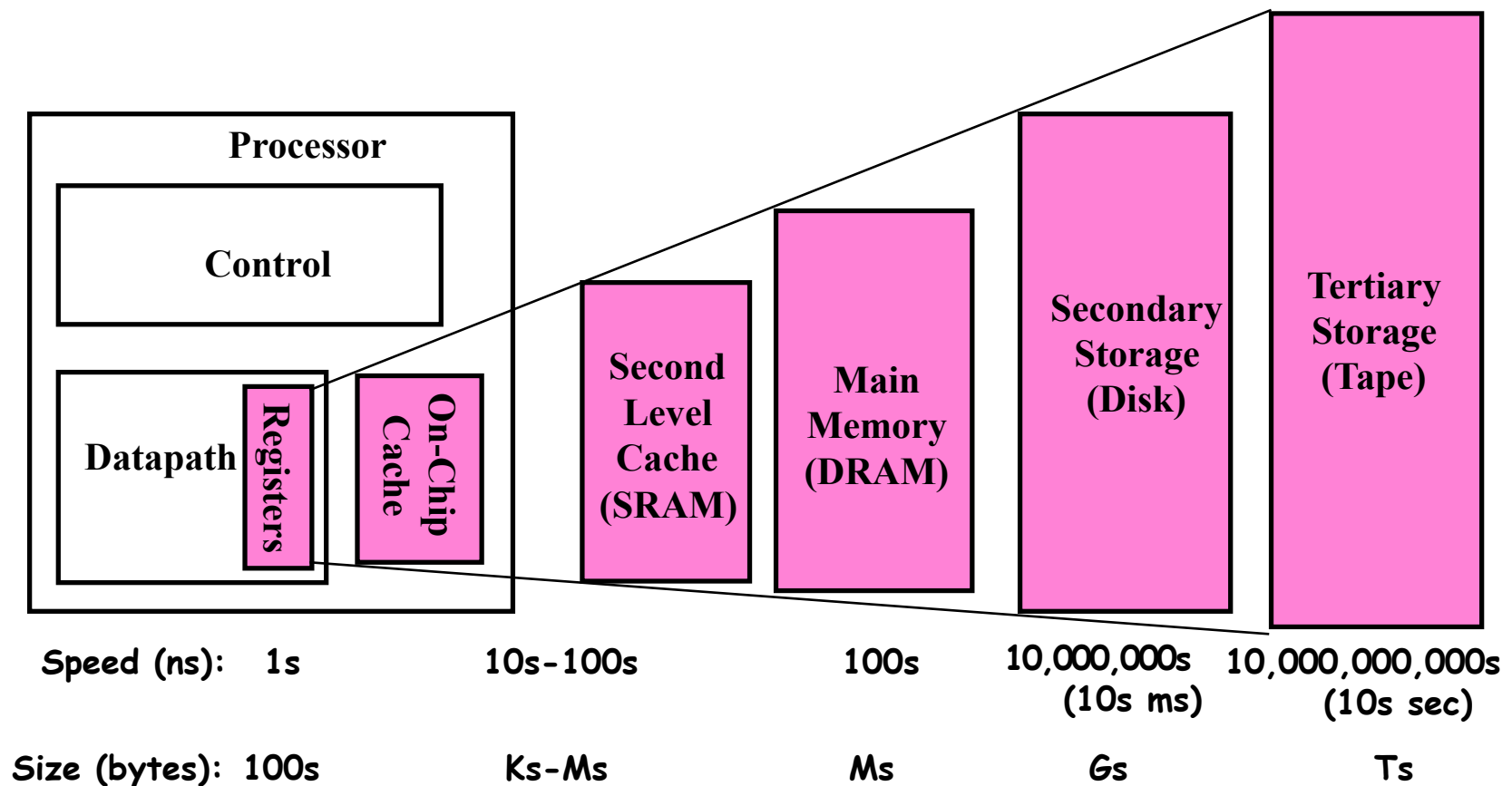


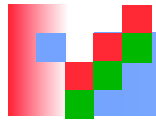
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Memory Hierarchy

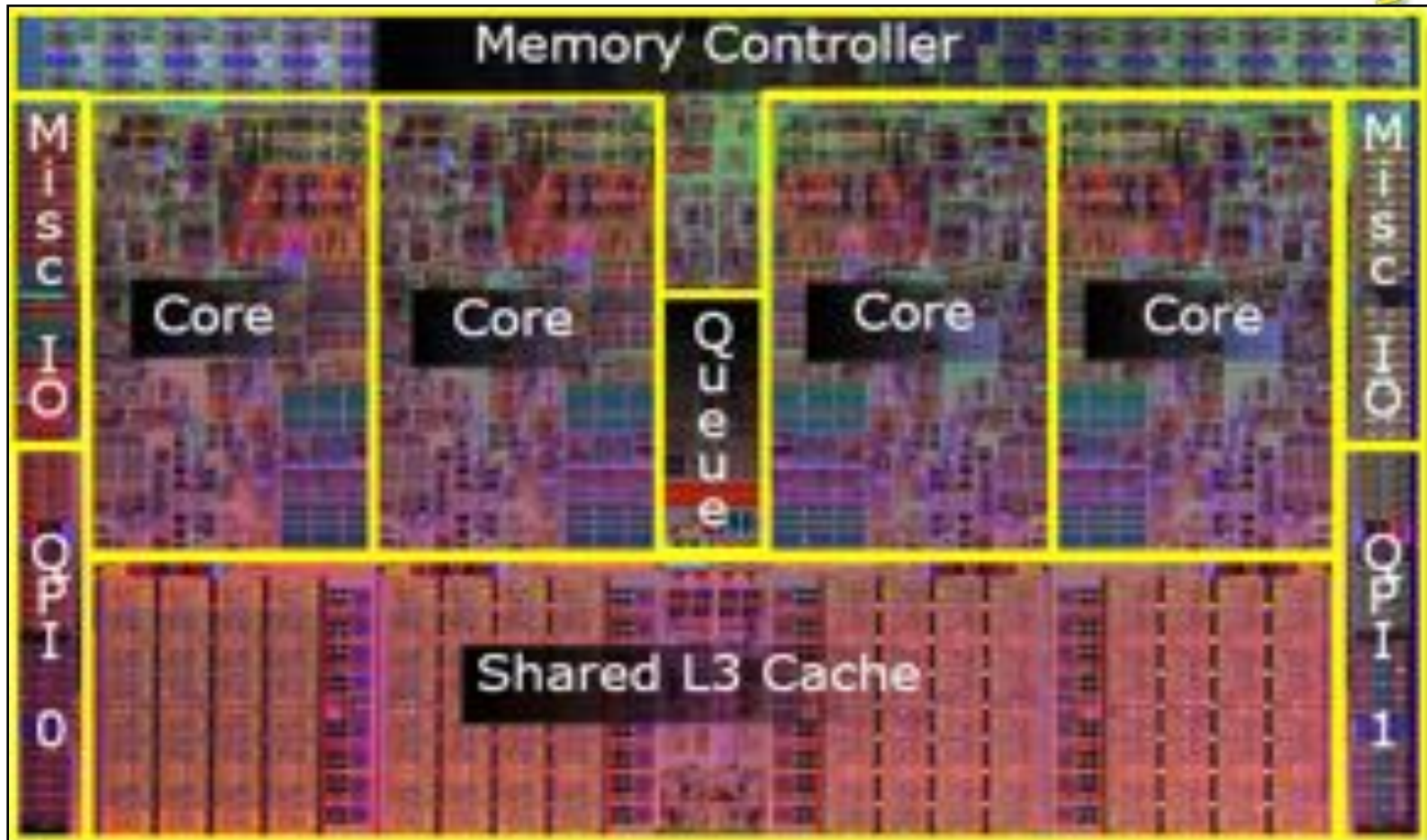
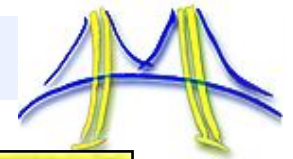


- Take advantage of the principle of locality to:
 - Present as much memory as in the cheapest technology
 - Provide access at speed offered by the fastest technology

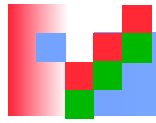




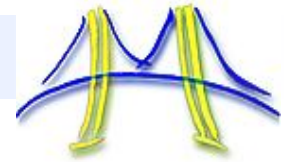
Example of modern core: Nehalem



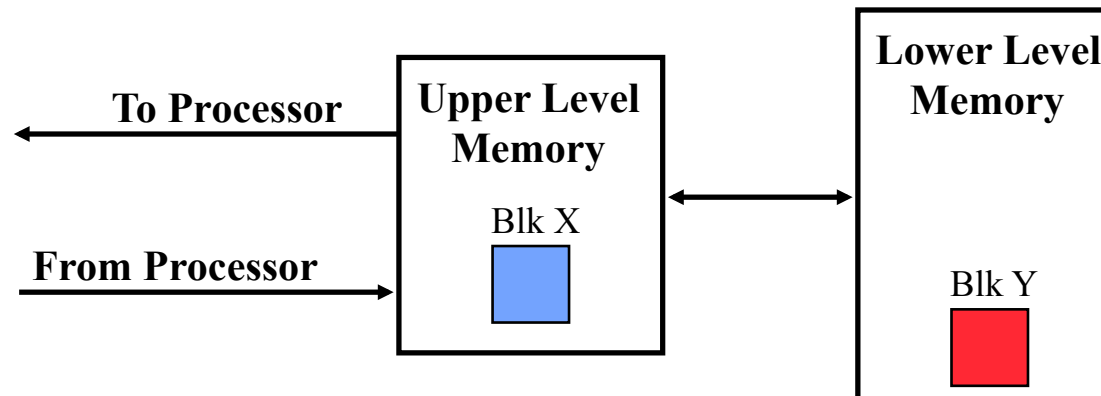
- ON-chip cache resources:
 - For each core: L1: 32K instruction and 32K data cache, L2: 1MB
 - L3: 8MB shared among all 4 cores
- Integrated, on-chip memory controller (DDR3)

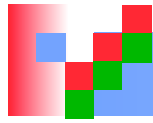


Memory Hierarchy: Terminology

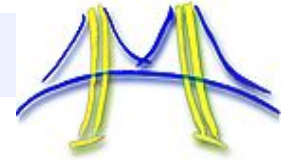


- **Hit**: data appears in some block in the upper level (example: Block X)
 - **Hit Rate**: the fraction of memory access found in the upper level
 - **Hit Time**: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- **Miss**: data needs to be retrieve from a block in the lower level (Block Y)
 - **Miss Rate** = $1 - (\text{Hit Rate})$
 - **Miss Penalty**: Time to replace a block in the upper level + Time to deliver the block the processor
- **Hit Time** \ll **Miss Penalty** (500 instructions on 21264!)

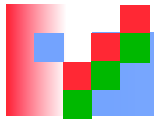




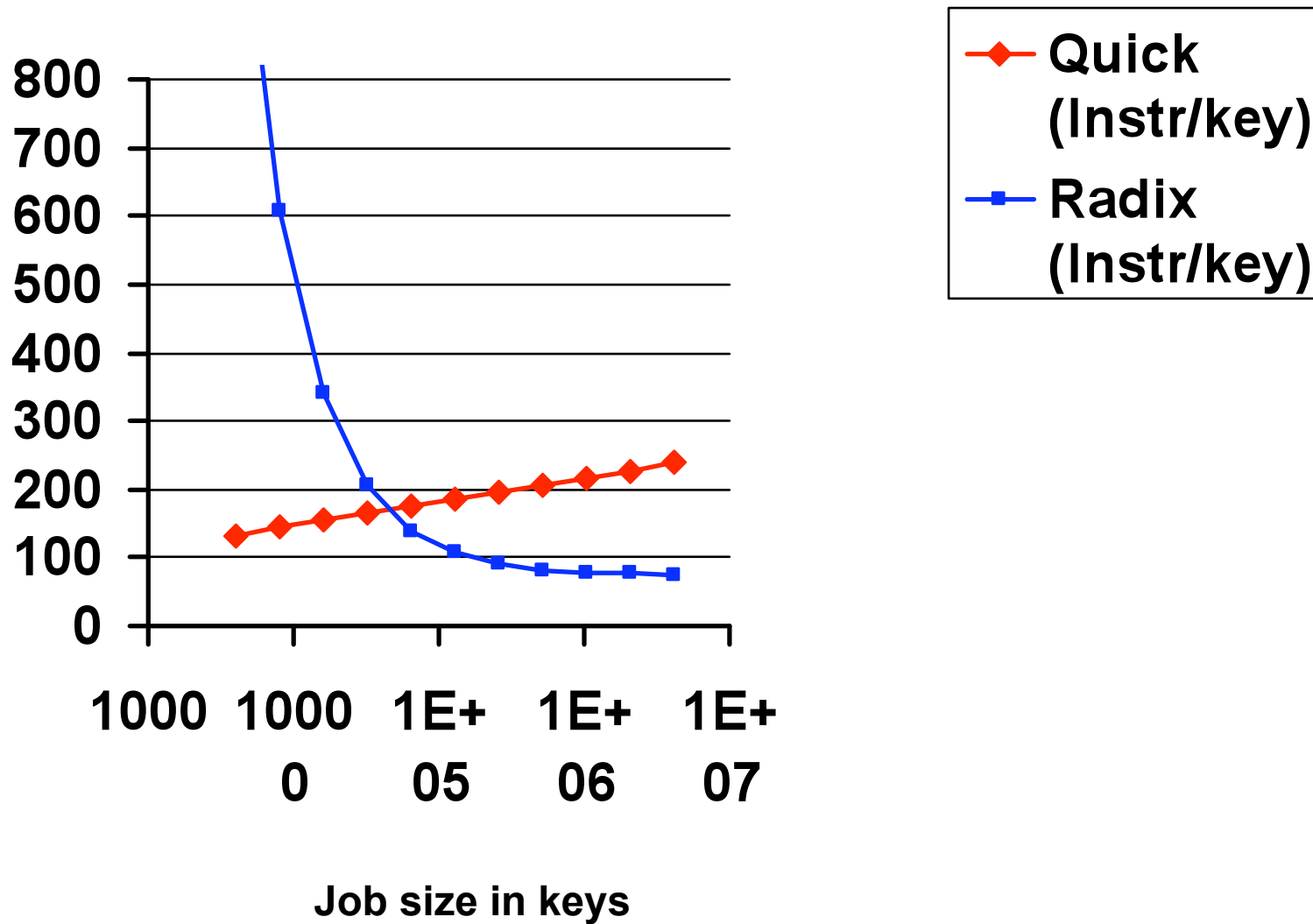
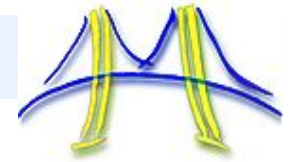
Impact of Hierarchy on Algorithms

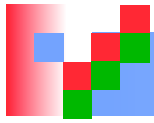


- Today CPU time is a function of (ops, cache misses)
- What does this mean to Compilers, Data structures, Algorithms?
 - Quicksort:
fastest comparison based sorting algorithm when keys fit in memory
 - Radix sort: also called "linear time" sort
For keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- "The Influence of Caches on the Performance of Sorting" by A. LaMarca and R.E. Ladner. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, January, 1997, 370-379.
 - For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000

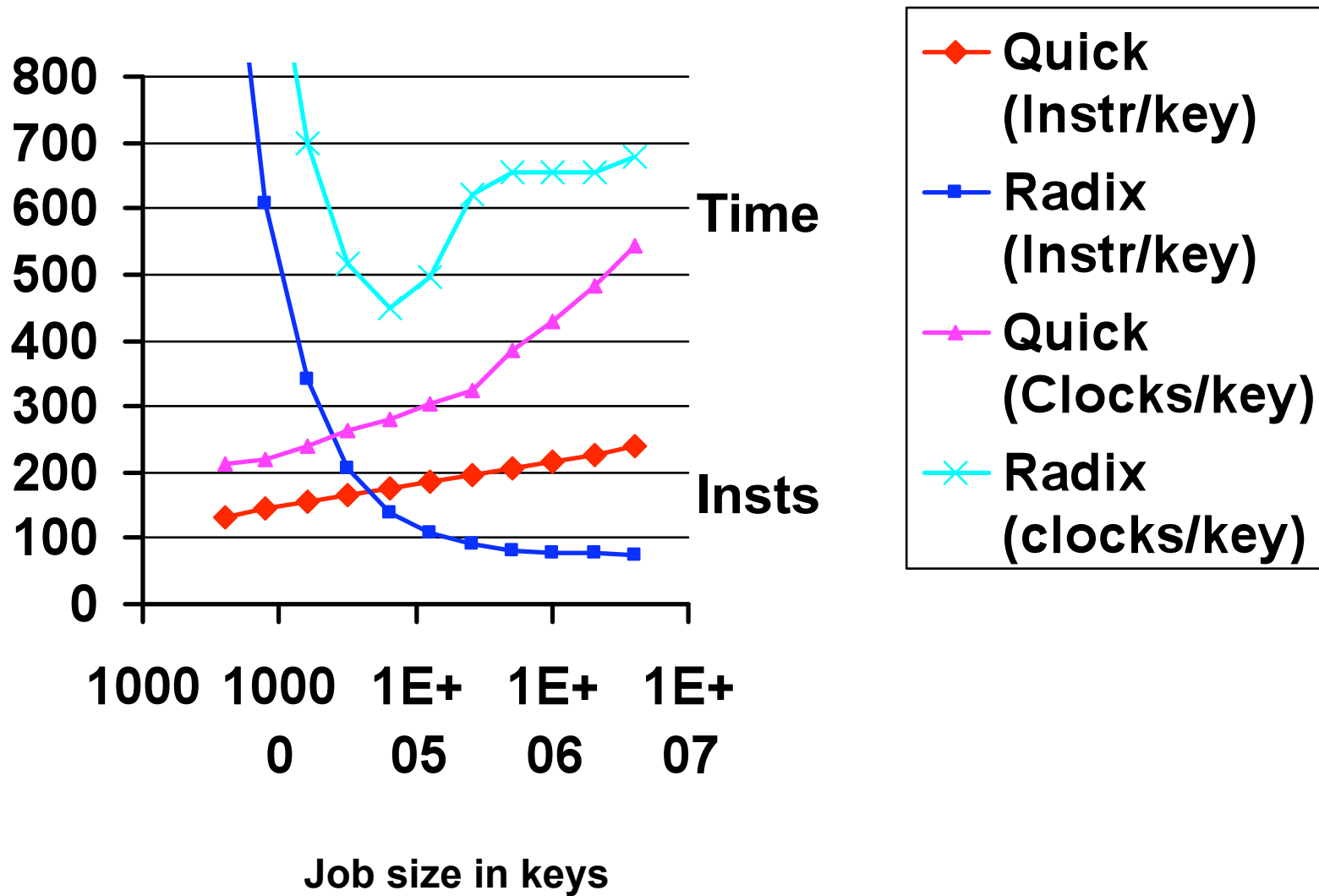
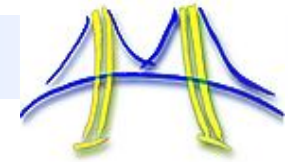


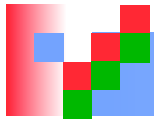
Quicksort vs. Radix: Instructions



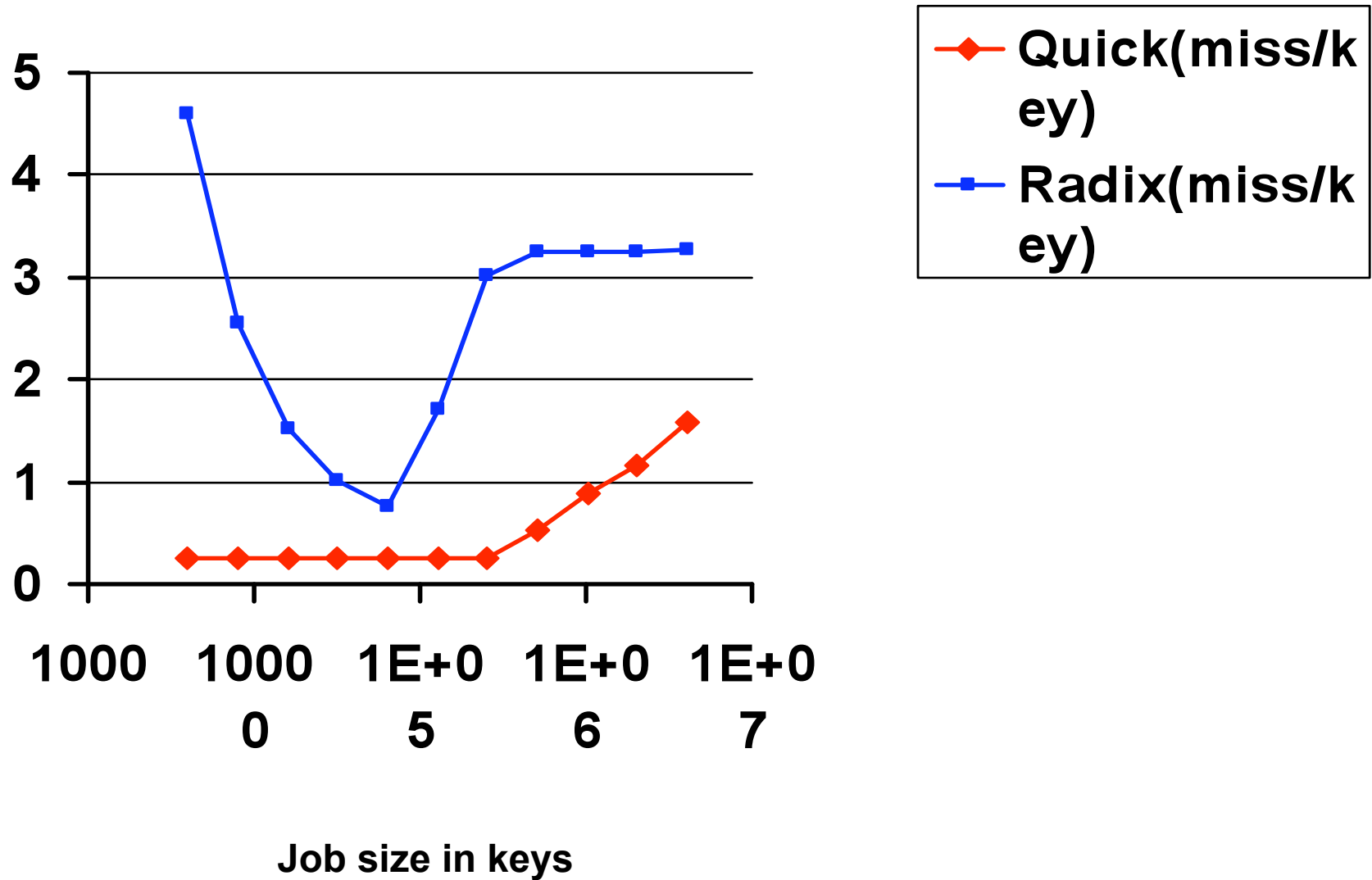
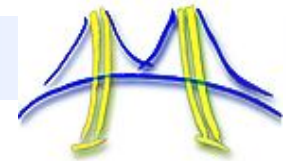


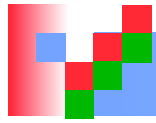
Quicksort vs. Radix Inst & Time



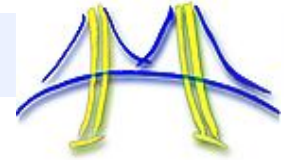


Quicksort vs. Radix: Cache misses

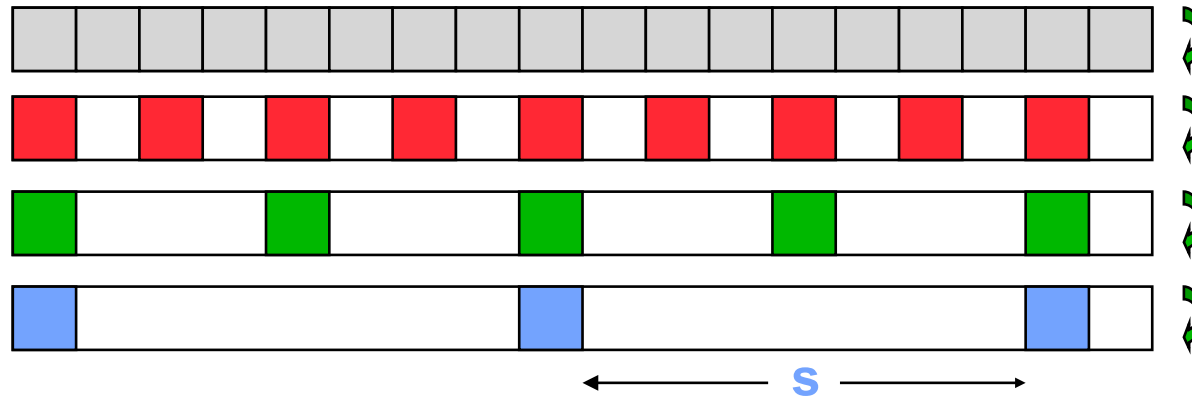




Experimental Study (Membench)

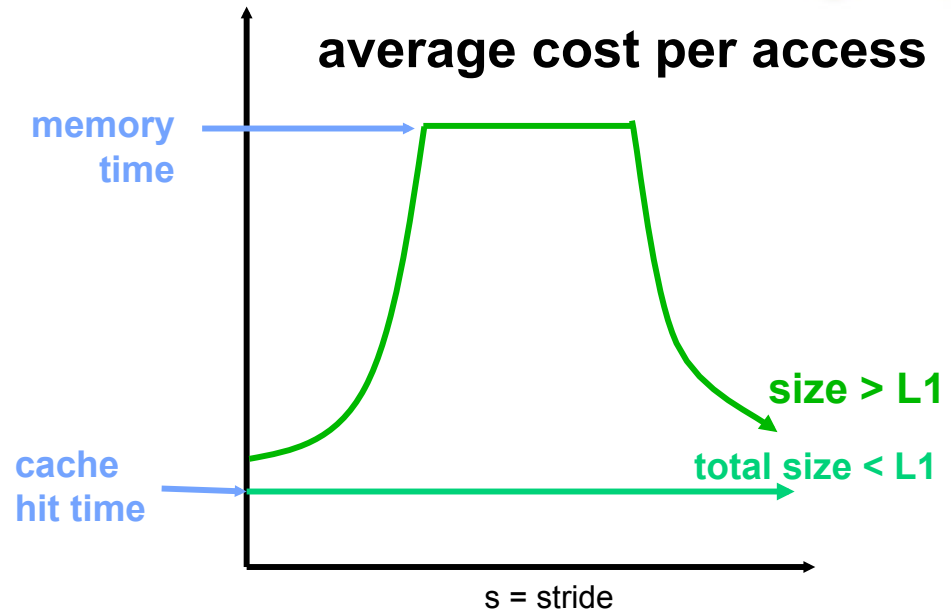
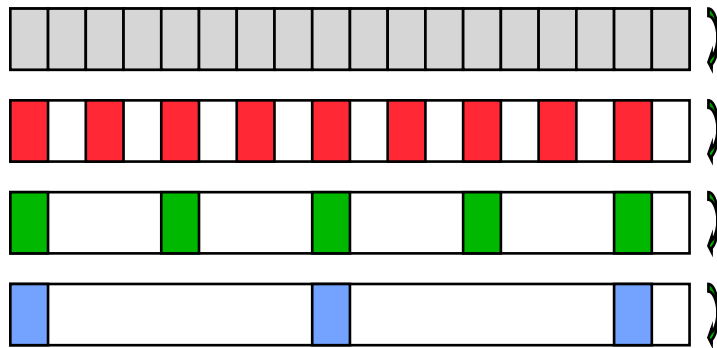
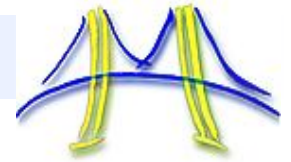


- Microbenchmark for memory system performance

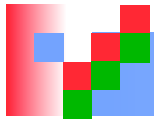


- for array A of length L from 4KB to 8MB by 2x
for stride s from 4 Bytes (1 word) to L/2 by 2x 1 experiment
time the following loop
(repeat many times and average)
for i from 0 to L by s
load A[i] from memory (4 Bytes)

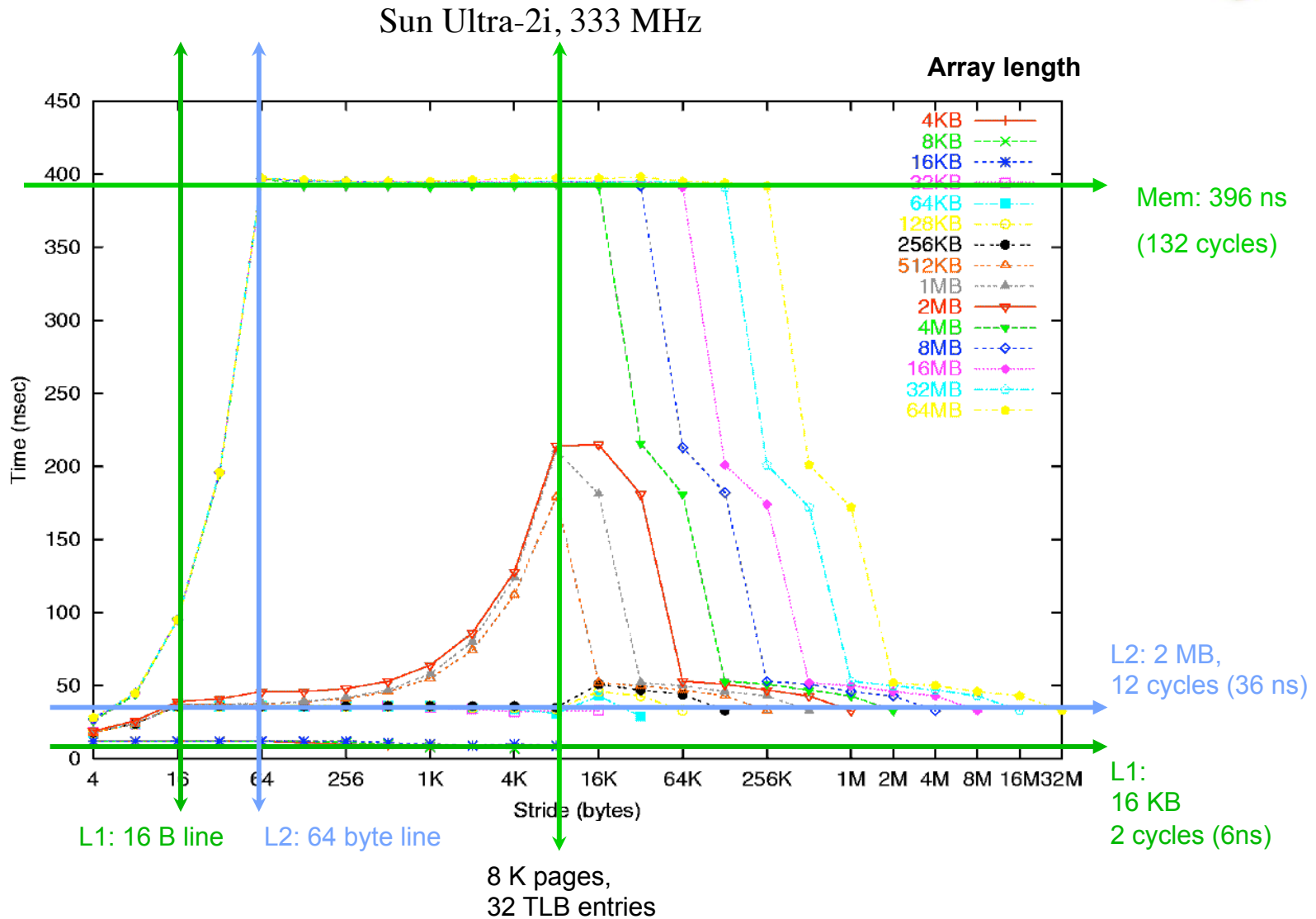
Membench: What to Expect



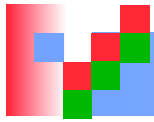
- Consider the average cost per load
 - Plot one line for each array length, time vs. stride
 - Small stride is best: if cache line holds 4 words, at most $\frac{1}{4}$ miss
 - If array is smaller than a given cache, all those accesses will hit (after the first run, which is negligible for large enough runs)
 - Picture assumes only one level of cache
 - Values have gotten more difficult to measure on modern procs



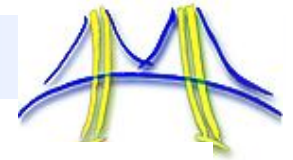
Memory Hierarchy on a Sun Ultra-2i



See www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps for details



Memory Hierarchy on a Power3

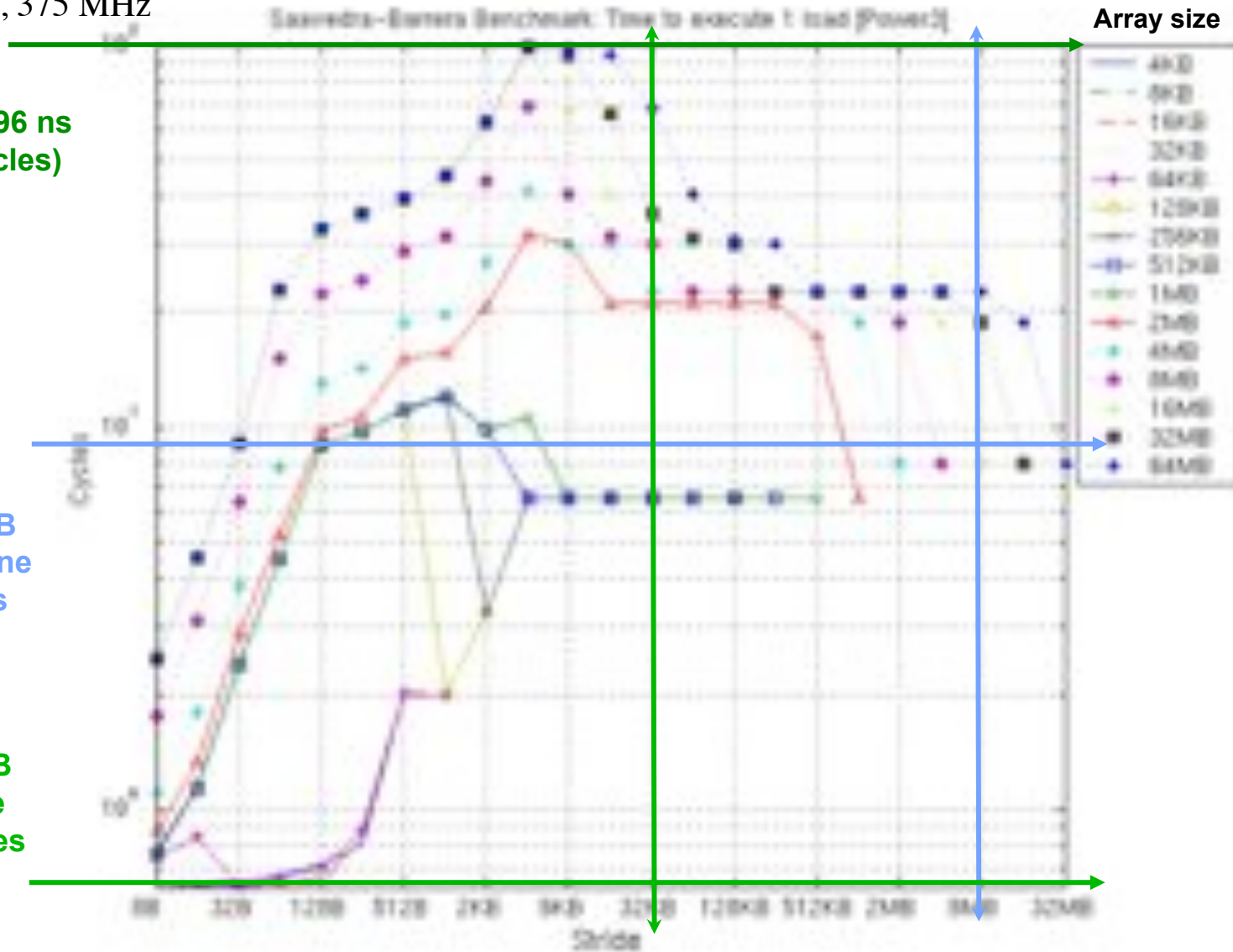


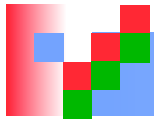
Power3, 375 MHz

Mem: 396 ns
(132 cycles)

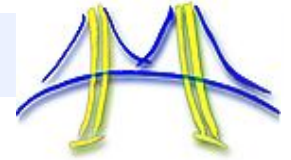
L2: 8 MB
128 B line
9 cycles

L1: 32 KB
128B line
.5-2 cycles

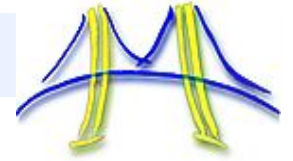
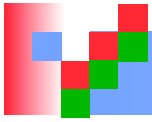




Memory Hierarchy Lessons



- Caches Vastly Impact Performance
 - Cannot consider performance without considering memory hierarchy
- Actual performance of a simple program can be a complicated function of the architecture
 - Slight changes in the architecture or program change the performance significantly
 - To write fast programs, need to consider architecture
 - » True on sequential or parallel processor
 - We would like simple models to help us design efficient algorithms
- Common technique for improving cache performance, called **blocking** or **tiling**:
 - Idea: used divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache
- **Autotuning**: Deal with complexity through experiments
 - Produce several different versions of code
 - » Different algorithms, Blocking Factors, Loop orderings, etc
 - For each architecture, run different versions to see which is fastest
 - Can (in principle) navigate complex design options for optimum



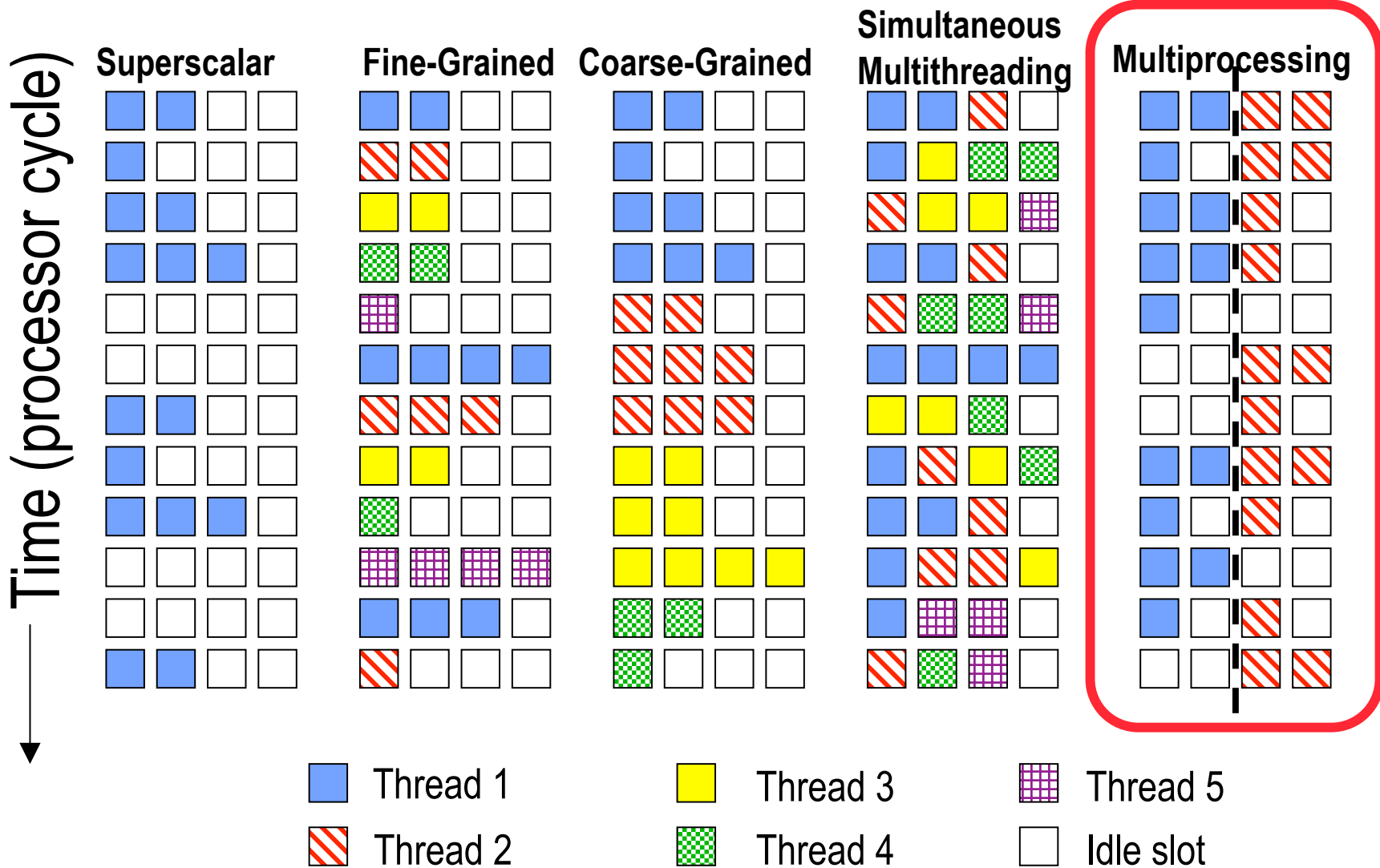
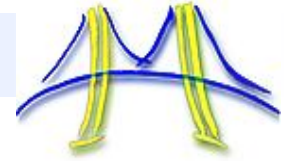
Explicitly Parallel Computer Architecture

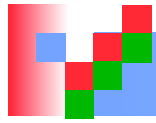
What is *Parallel Architecture*?



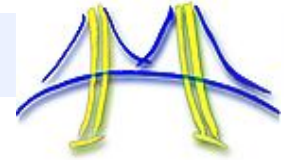
- A parallel computer is a collection of processing elements that cooperate to solve large problems
 - Most important new element: It is all about communication!
- What does the programmer (or OS or Compiler writer) think about?
 - Models of computation:
 - » PRAM? BSP? Sequential Consistency?
 - Resource Allocation:
 - » how powerful are the elements?
 - » how much memory?
- What mechanisms must be in hardware vs software
 - What does a single processor look like?
 - » High performance general purpose processor
 - » SIMD processor
 - » Vector Processor
 - Data access, Communication and Synchronization
 - » how do the elements cooperate and communicate?
 - » how are data transmitted between processors?
 - » what are the abstractions and primitives for cooperation?

Types of Parallelism

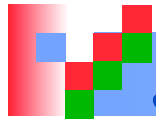




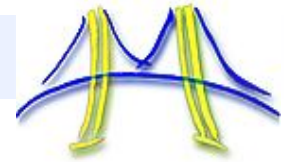
Parallel Programming Models



- **Programming model** is made up of the languages and libraries that create an abstract view of the machine
- **Control**
 - How is parallelism **created**?
 - What **orderings** exist between operations?
 - How do different threads of control **synchronize**?
- **Data**
 - What data is **private** vs. **shared**?
 - How is logically shared data accessed or **communicated**?
- **Synchronization**
 - What operations can be used to coordinate parallelism
 - What are the **atomic** (indivisible) operations?
- **Cost**
 - How do we account for the **cost** of each of the above?



Simple Programming Example



- Consider applying a function f to the elements of an array A and then computing its sum:

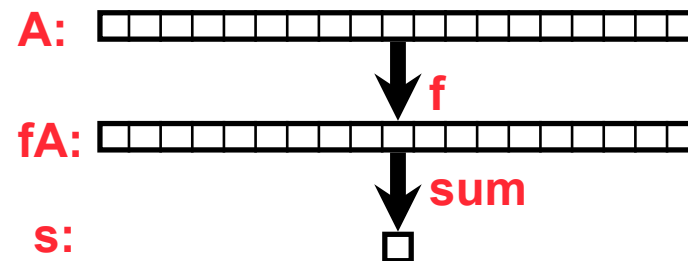
$$\sum_{i=0}^{n-1} f(A[i])$$

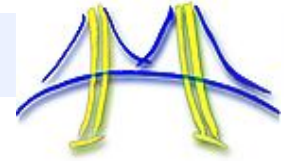
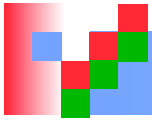
- Questions:
 - Where does A live? All in single memory? Partitioned?
 - What work will be done by each processors?
 - They need to coordinate to get a single result, how?

A = array of all data

$fA = f(A)$

$s = \text{sum}(fA)$



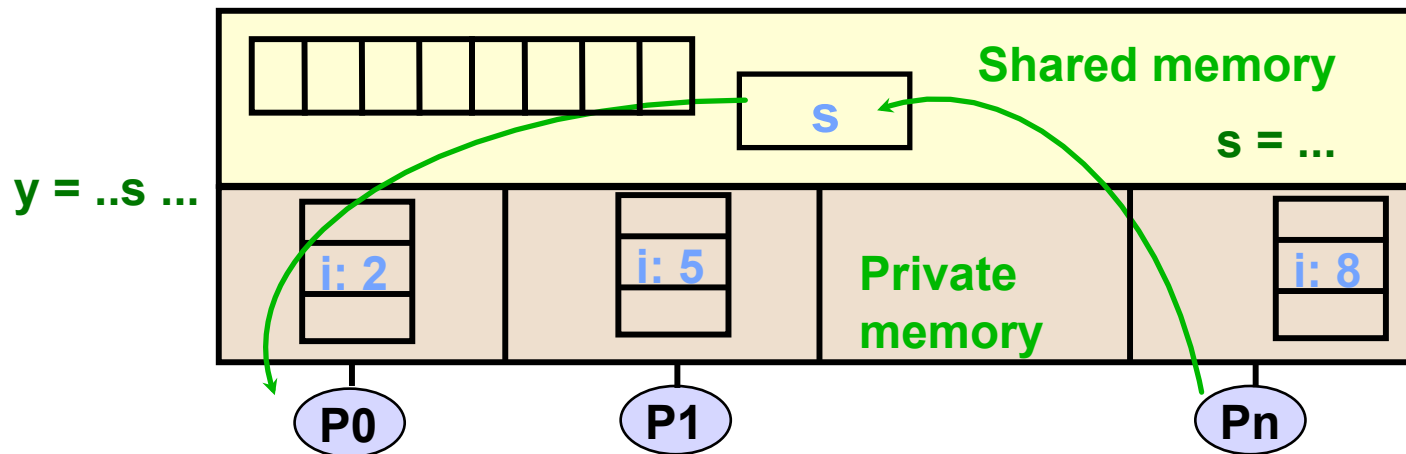


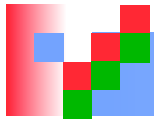
Shared Memory Programming Model

Programming Model 1: Shared Memory

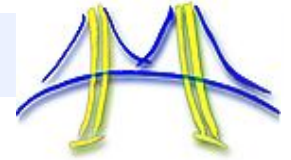


- Program is a collection of threads of control.
 - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of **private variables**, e.g., local stack variables
- Also a set of **shared variables**, e.g., static variables, shared common blocks, or global heap.
 - Threads communicate **implicitly** by writing and reading shared variables.
 - Threads coordinate by **synchronizing** on shared variables





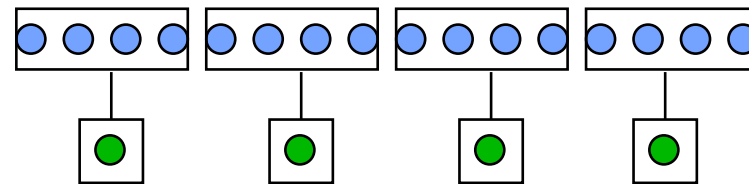
Simple Programming Example: SM



- Shared memory strategy:
 - small number $p \ll n = \text{size}(A)$ processors
 - attached to single memory

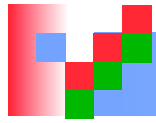
$$\sum_{i=0}^{n-1} f(A[i])$$

- Parallel Decomposition:
 - Each evaluation and each partial sum is a task.
- Assign n/p numbers to each of p procs
 - Each computes independent "private" results and partial sum.
 - Collect the p partial sums and compute a global sum.

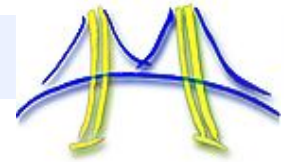


Two Classes of Data:

- Logically Shared
 - The original n numbers, the global sum.
- Logically Private
 - The individual function evaluations.
 - What about the individual partial sums?



Shared Memory "Code" for sum



```
static int s = 0;
```

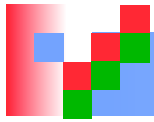
Thread 1

```
for i = 0, n/2-1  
  s = s + f(A[i])
```

Thread 2

```
for i = n/2, n-1  
  s = s + f(A[i])
```

- Problem is a race condition on variable s in the program
- A **race condition** or **data race** occurs when:
 - two processors (or two threads) access the same variable, and at least one does a write.
 - The accesses are concurrent (not synchronized) so they could happen simultaneously



Better Shared Memory Code for Sum?



A

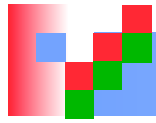
3	5
---	---

 f = square

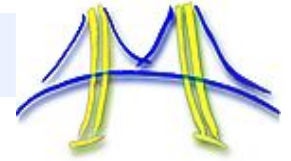
```
static int s = 0;
```

Thread 1	Thread 2
....	...
compute f([A[i]) and put in reg0 9	compute f([A[i]) and put in reg0 25
reg1 = s 0	reg1 = s 0
reg1 = reg1 + reg0 9	reg1 = reg1 + reg0 25
s = reg1 9	s = reg1 25
...	...

- Assume A = [3,5], f is the square function, and s=0 initially
- For this program to work, s should be 34 at the end
 - but it may be 34,9, or 25
- The *atomic* operations are reads and writes
 - Never see 1/2 of one number, but += operation is not atomic
 - All computations happen in (private) registers



Improved Code for Sum



```
static int s = 0;  
static lock lk;
```

Thread 1

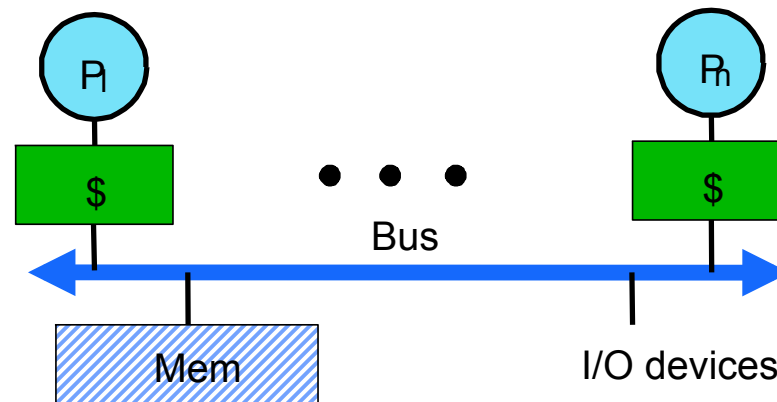
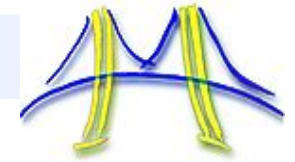
```
local_s1 = 0  
for i = 0, n/2-1  
    local_s1 = local_s1 + f(A[i])  
lock(lk);  
s = s + local_s1  
unlock(lk);
```

Thread 2

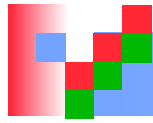
```
local_s2 = 0  
for i = n/2, n-1  
    local_s2 = local_s2 + f(A[i])  
lock(lk);  
s = s + local_s2  
unlock(lk);
```

- Since addition is associative, it's OK to rearrange order
- Most computation is on private variables
 - Sharing frequency is also reduced, which might improve speed
 - But there is still a race condition on the update of shared s
- The race condition can be fixed by adding locks (only one thread can hold a lock at a time; others wait for it)

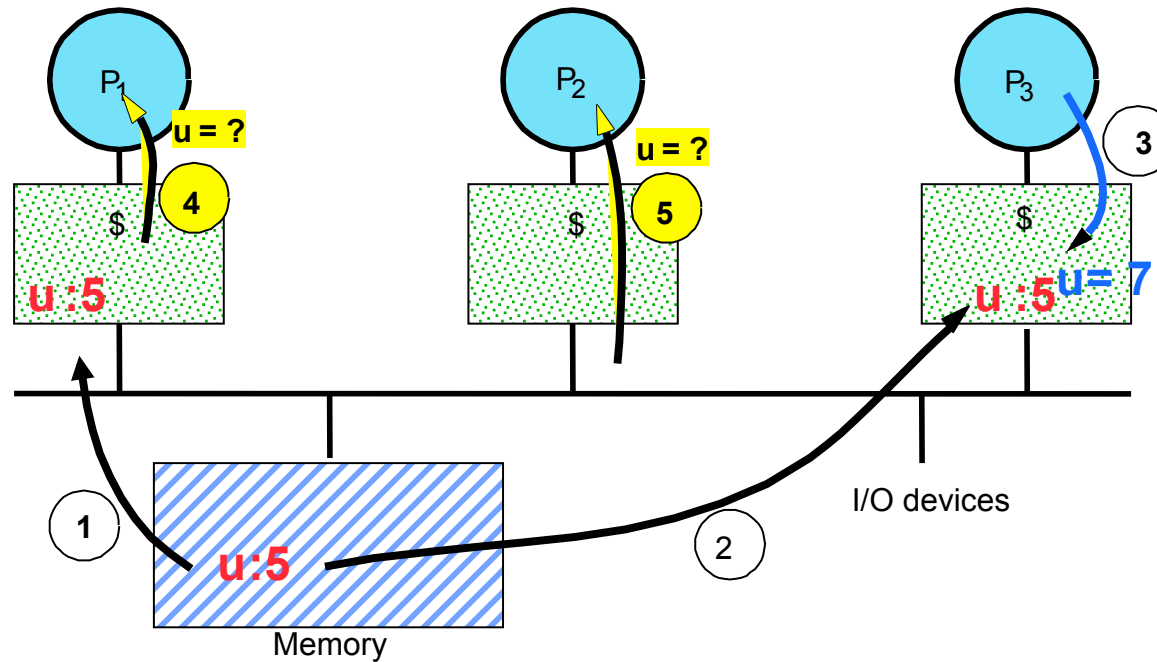
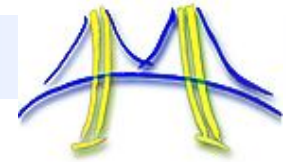
What About Caching???



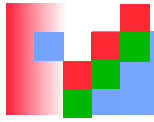
- Want High performance for shared memory: Use Caches!
 - Each processor has its own cache (or multiple caches)
 - Place data from memory into cache
 - Writeback cache: don't send all writes over bus to memory
- Caches Reduce average latency
 - Automatic replication closer to processor
 - *More* important to multiprocessor than uniprocessor: latencies longer
- Normal uniprocessor mechanisms to access data
 - Loads and Stores form very low-overhead communication primitive
- **Problem: Cache Coherence!**



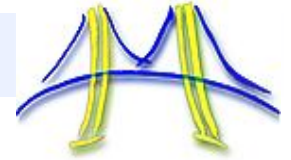
Example Cache Coherence Problem



- Things to note:
 - Processors could see different values for u after event 3
 - With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
- How to fix with a bus: Coherence Protocol
 - Use bus to broadcast writes or invalidations
 - Simple protocols rely on presence of broadcast medium
- Bus not scalable beyond about 64 processors (max)
 - Capacitance, bandwidth limitations



Example: Coherence not Enough



P₁

P₂

*/*Assume initial value of A and ag is 0*/*

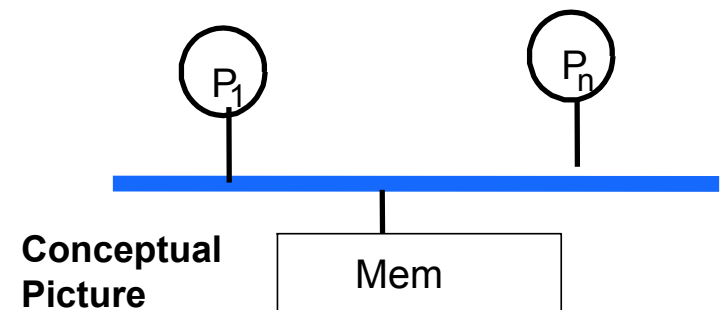
A = 1;

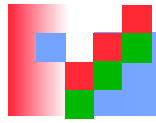
while (flag == 0); */*spin idly*/*

flag = 1;

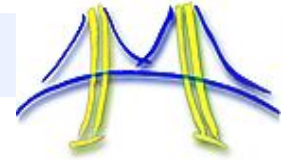
print A;

- Intuition not guaranteed by coherence
- expect memory to respect order between accesses to *different* locations issued by a given process
 - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
 - pertains only to single location
 - Need statement about ordering between multiple locations.

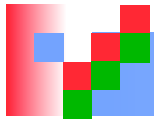




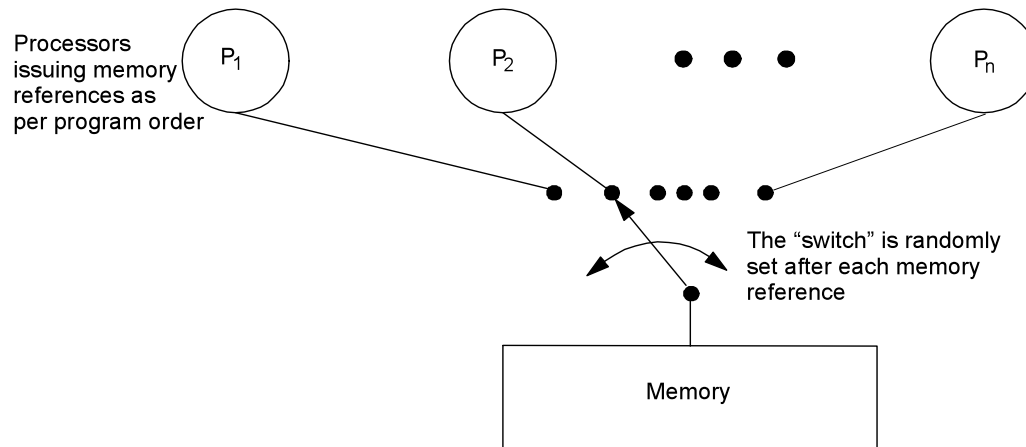
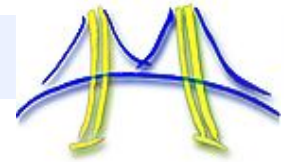
Memory Consistency Model



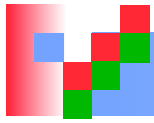
- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
 - What orders are preserved?
 - Given a load, constrains the possible values returned by it
- Without it, can't tell much about a single address space (SAS) program's execution
- Implications for both programmer and system designer
 - Programmer uses to reason about correctness and possible results
 - System designer can use to constrain how much accesses can be reordered by compiler or hardware
- Contract between programmer and system



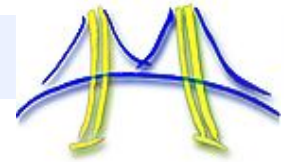
Sequential Consistency



- Total order achieved by interleaving accesses from different processes
 - Maintains program order, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
 - as if there were no caches, and a single memory
- "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."
[Lamport, 1979]



Sequential Consistency Example



Processor 1

LD₁ A ⇒ 5
LD₂ B ⇒ 7
ST₁ A, 6
...
LD₃ A ⇒ 6
LD₄ B ⇒ 21
ST₂ B, 13
ST₃ B, 4

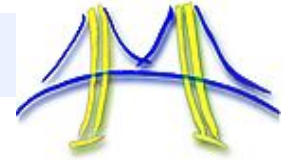
Processor 2

LD₅ B ⇒ 2
...
LD₆ A ⇒ 6
ST₄ B, 21
...
LD₇ A ⇒ 6
...
LD₈ B ⇒ 4

One Consistent Serial Order

LD₁ A ⇒ 5
LD₂ B ⇒ 7
LD₅ B ⇒ 2
ST₁ A, 6
LD₆ A ⇒ 6
ST₄ B, 21
LD₃ A ⇒ 6
LD₄ B ⇒ 21
LD₇ A ⇒ 6
ST₂ B, 13
ST₃ B, 4
LD₈ B ⇒ 4

What about Synchronization?

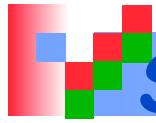


- All shared-memory programs need synchronization
 - Problem: Communication is *IMPLICIT* thus, no way of knowing when other threads have completed their operations
 - Consider need for "lock" primitive in previous example
 - Barrier - global (/coordinated) synchronization
 - simple use of barriers -- all threads hit the same one

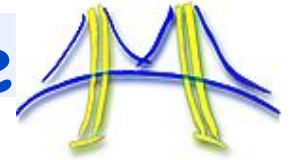
```
work_on_my_subgrid();  
barrier;  
read_neighboring_values();  
barrier;
```

 - barriers are not provided in all thread libraries
 - Mutexes - mutual exclusion locks
 - threads are mostly independent and must access common data

```
lock *l = alloc_and_init();    /* shared */  
lock(l);  
    access data  
unlock(l);
```
- Another Option: Transactional memory
 - Hardware equivalent of optimistic concurrency
 - Some think that this is the answer to all parallel programming



Synchronization using load and store



- Here is a possible two-thread synchronization:

Thread A

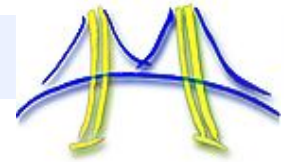
```
Set LockA=1;  
while (LockB) { //X  
    do nothing;  
}  
Critical Section;  
Set LockA=0;
```

Thread B

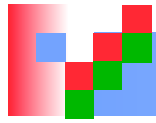
```
Set LockB=1;  
if (!LockA) { //Y  
    Critical Section;  
}  
Set LockB=0;
```

- Does this work? **Yes**. Both can guarantee that:
 - Only one will enter critical section at a time.
- At X:
 - if Lock_B=0, safe for A to perform critical section,
 - otherwise wait to find out what will happen
- At Y:
 - if Lock_A=0, safe for B to perform critical section.
 - Otherwise, A is in critical section or waiting for B to quit
- But:
 - Really messy
 - Generalization gets worse
 - **Needs Sequential Consistency to work!**

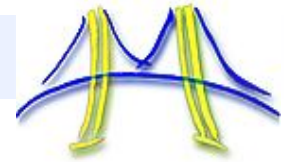
Need Hardware Atomic Primitives



- ```
test&set (&address) { /* most architectures */
 result = M[address];
 M[address] = 1;
 return result;
}
```
- ```
swap (&address, register) { /* x86 */
    temp = M[address];
    M[address] = register;
    register = temp;
}
```
- ```
compare&swap (&address, reg1, reg2) { /* 68000 */
 if (reg1 == M[address]) {
 M[address] = reg2;
 return success;
 } else {
 return failure;
 }
}
```
- ```
load-linked&store conditional (&address) {
    /* R4000, alpha */
    loop:
        ll r1, M[address];
        movi r2, 1; /* Can do arbitrary comp */
        sc r2, M[address];
        beqz r2, loop;
}
```



Implementing Locks with test&set



- A flawed, but simple solution:

```
int value = 0; // Free
Acquire() {
    while (test&set(value)); // while busy
}
Release() {
    value = 0;
}
```

- Simple explanation:

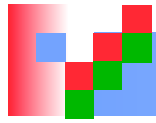
- If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
- If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
- When we set value = 0, someone else can get lock

- Problems:

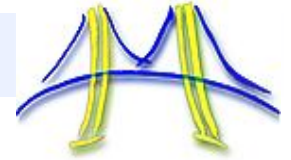
- **Busy-Waiting**: thread consumes cycles while waiting
- **Unfair**: may give advantage to some processors over others
- **Expensive**: Every test&set() for every processor goes across network!

- Better: test&test&set

- Use outer loop that only reads value, watches for value=0

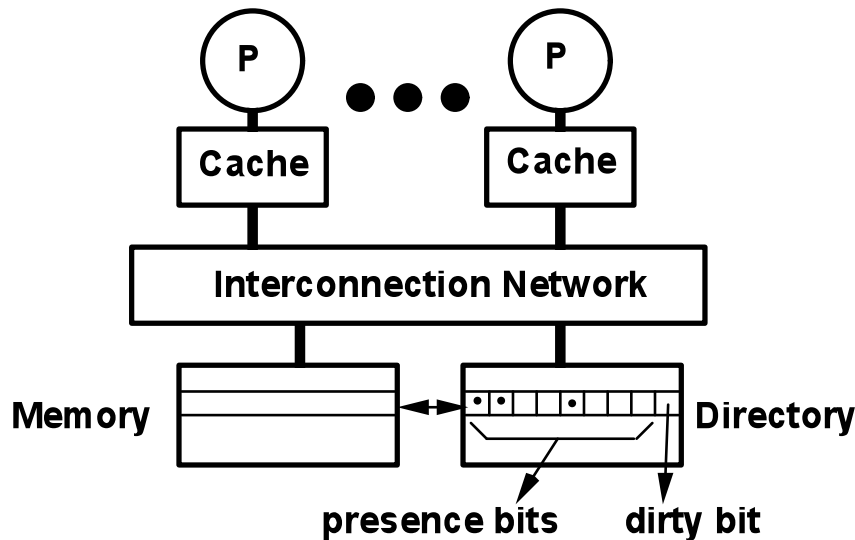
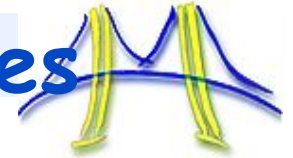


Busy-wait vs Blocking



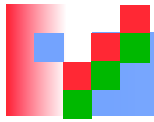
- Busy-wait: I.e. spin lock
 - Keep trying to acquire lock until read
 - Very low latency/processor overhead!
 - Very high system overhead!
 - » Causing stress on network while spinning
 - » Processor is not doing anything else useful
- Blocking:
 - If can't acquire lock, deschedule process (I.e. unload state)
 - Higher latency/processor overhead (1000s of cycles?)
 - » Takes time to unload/restart task
 - » Notification mechanism needed
 - Low system overhead
 - » No stress on network
 - » Processor does something useful
- Hybrid:
 - Spin for a while, then block
 - 2-competitive: spin until have waited blocking time

Scalable Shared Memory: Directories

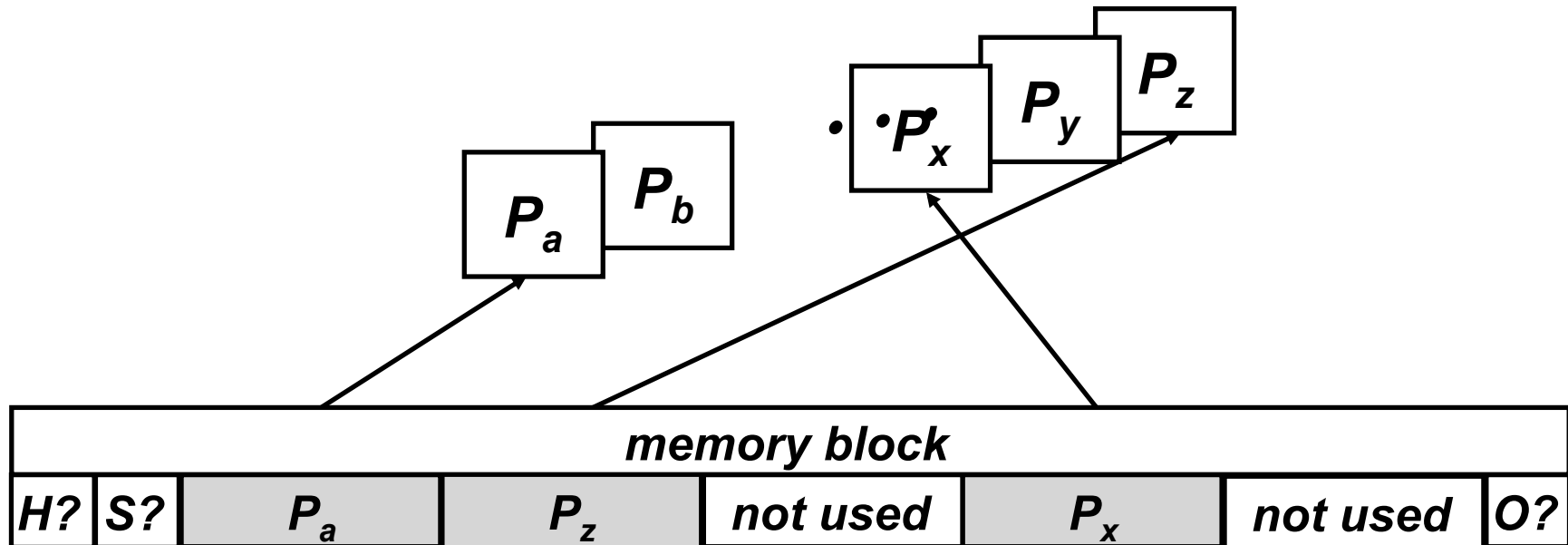
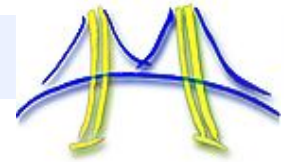


- k processors.
- With each cache-block in memory: k presence-bits, 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit

- Every memory block has associated directory information
 - keeps track of copies of cached blocks and their states
 - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
 - in scalable networks, communication with directory and copies is through network transactions
- Each Reader recorded in directory
- Processor asks permission of memory before writing:
 - Send invalidation to each cache with read-only copy
 - Wait for acknowledgements before returning permission for writes

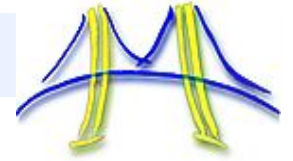
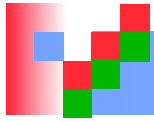


Reducing the Directory Size: "Limitless directories" (Alewife, MIT)

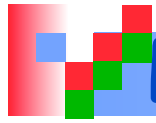


Instead of a N -bit-vector, keep n ($\lg N$ -bit) pointers;
if more than n children request a copy, handle the
overflow in software

effective for large N and low degree of sharing



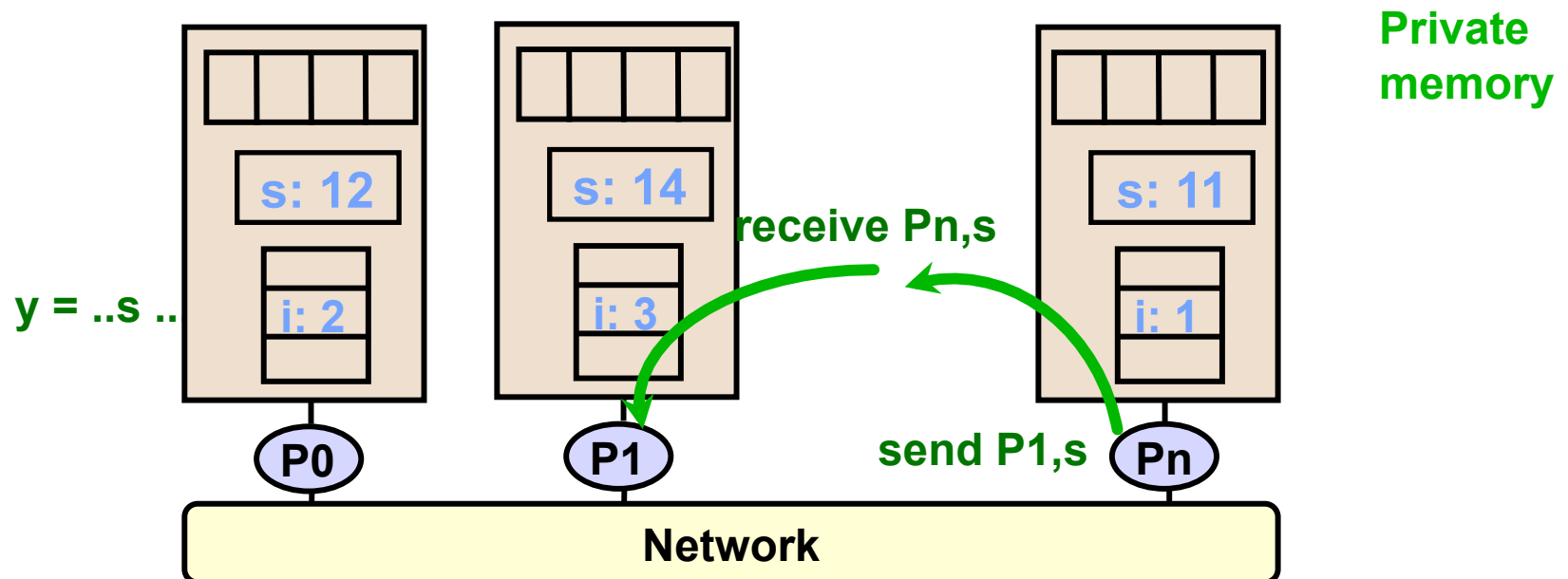
Message Passing Programming Model

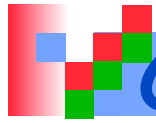


Programming Model 2: Message Passing

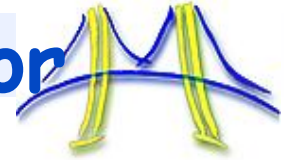


- Program consists of a collection of **named** processes.
 - Usually fixed at program startup time
 - Thread of control plus local address space -- NO shared data.
 - Logically shared data is partitioned over local processes.
- Processes communicate by explicit send/receive pairs
 - Coordination is implicit in every communication event.
 - MPI (Message Passing Interface) is the most commonly used SW





Compute $A[1]+A[2]$ on each processor



- First possible solution – what could go wrong?

Processor 1
xlocal = A[1]
send xlocal, proc2
receive xremote, proc2
s = xlocal + xremote

Processor 2
xlocal = A[2]
send xlocal, proc1
receive xremote, proc1
s = xlocal + xremote

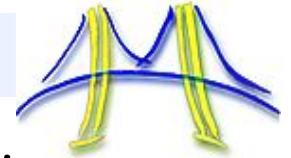
- If send/receive acts like the telephone system? The post office?
- Second possible solution

Processor 1
xlocal = A[1]
send xlocal, proc2
receive xremote, proc2
s = xlocal + xremote

Processor 2
xloadl = A[2]
receive xremote, proc1
send xlocal, proc1
s = xlocal + xremote

- What if there are more than 2 processors?

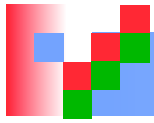
MPI - the de facto standard



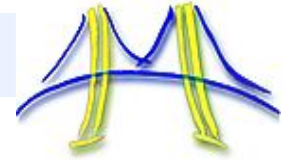
- MPI has become the de facto standard for parallel computing using message passing
- Example:

```
for (i=1; i<numprocs; i++) {
    sprintf(buff, "Hello %d! ", i);
    MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG,
             MPI_COMM_WORLD);
}
for (i=1; i<numprocs; i++) {
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG,
            MPI_COMM_WORLD, &stat);
    printf("%d: %s\n", myid, buff);
}
```

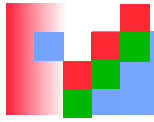
- Pros and Cons of standards
 - MPI created finally a standard for applications development in the HPC community → portability
 - The MPI standard is a least common denominator building on mid-80s technology, so may discourage innovation



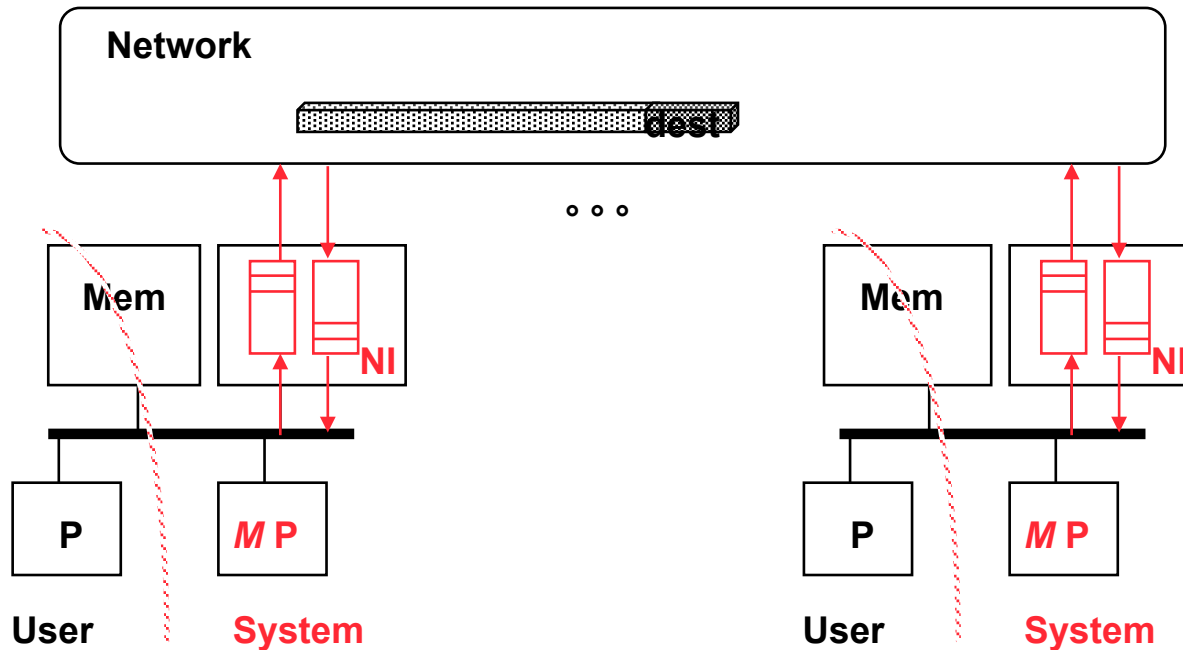
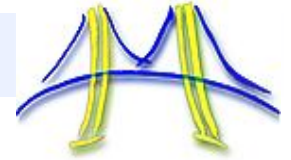
Message Passing Details



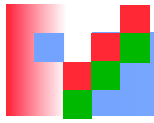
- All data layout must be handled by software
 - cannot retrieve remote data except with message request/reply
 - Often, message passing code produced by a compiler
- Message passing has high software overhead
 - early machines had to invoke OS on each message (100 μ s-1ms/message)
 - even user level access to network interface has dozens of cycles overhead (NI might be on I/O bus)
 - sending can be cheap (just like stores), but requires HW support
 - » Still requires some sort of *marshalling* of data into message
 - receiving is often expensive without special HW:
 - » need to poll or deal with an interrupt
- Active Message Abstraction
 - Message contains handler that is automatically invoked at destination
 - Can be utilized to support dataflow in hardware
 - Can be utilized to efficiently support compiled dataflow languages
 - » i.e. Id \rightarrow TAM as shown by Culler et al.
 - Can also serve as good target for compiled Shared-Address Space programs running on message passing hardware
 - » i.e UPC produced by Titanium when compiling apps (Yellick et al.)



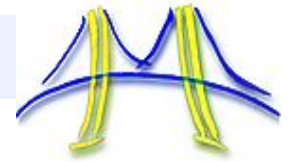
Dedicated Message Processor



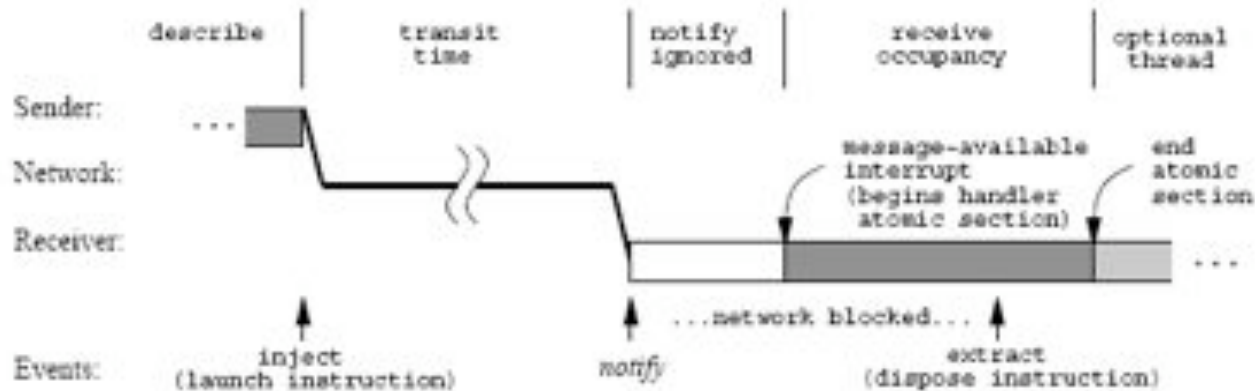
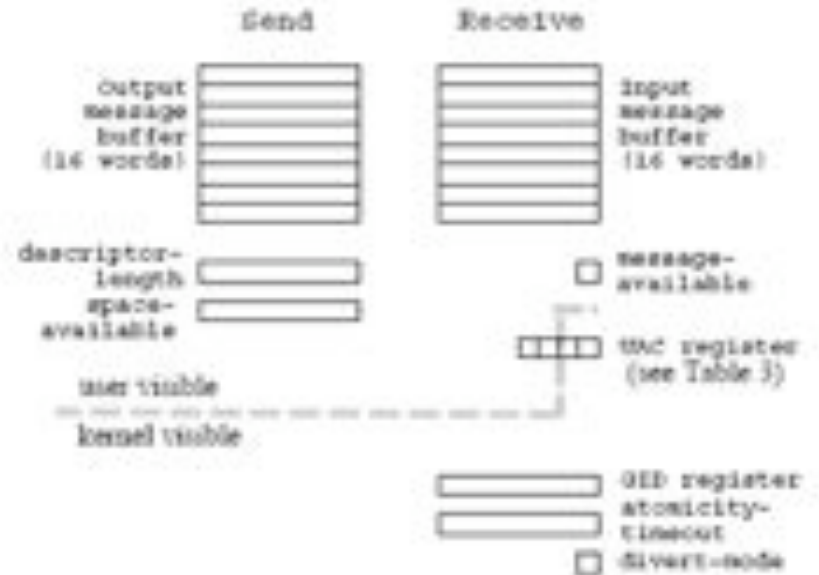
- General Purpose processor performs arbitrary output processing (at system level)
- General Purpose processor interprets incoming network transactions (at system level)
- User Processor \leftrightarrow Msg Processor share memory
- Msg Processor \leftrightarrow Msg Processor via system network transaction

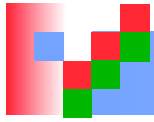


Asynchronous User-Level Networking (Alewife)

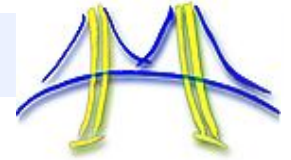


- Send message
 - write words to special network interface registers
 - Execute atomic launch instruction
- Receive
 - Generate interrupt/launch user-level thread context
 - Examine message by reading from special network interface registers
 - Execute dispose message
 - Exit atomic section

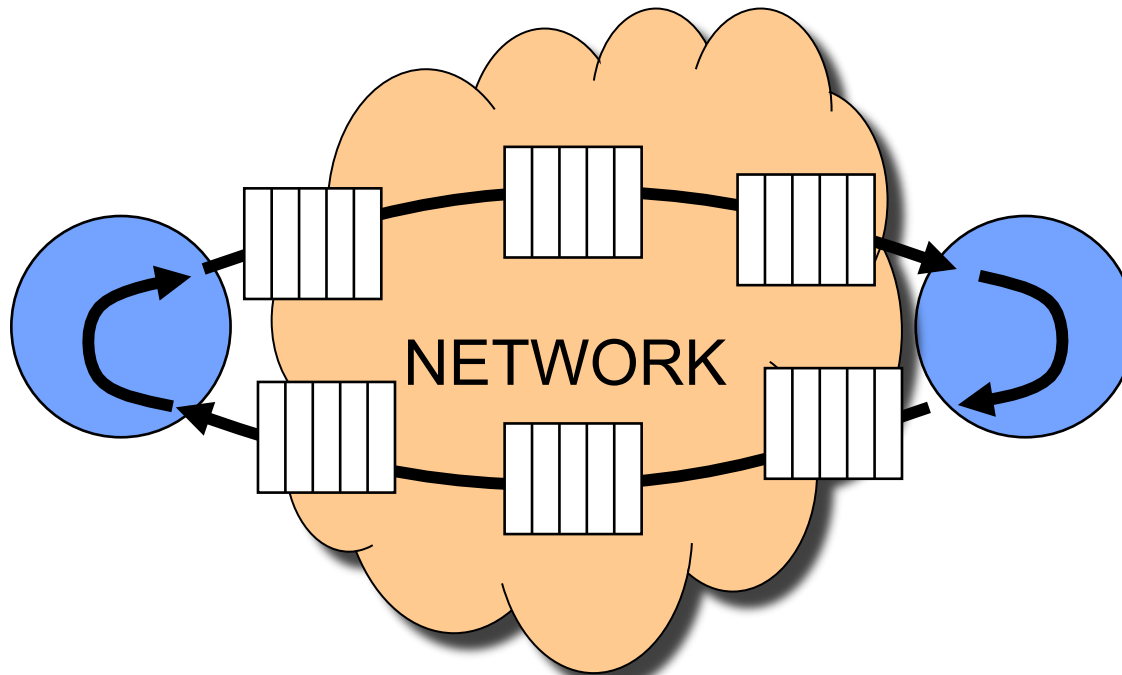




Danger of direct access to network: The Fetch Deadlock Problem



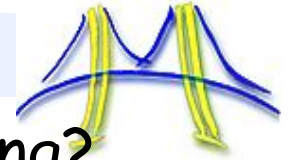
- Even if a node cannot issue a request, it must sink network transactions!
 - Incoming transaction may be request \Rightarrow generate a response.
 - Closed system (finite buffering)
- Deadlock occurs even if network deadlock free!



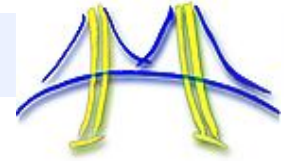
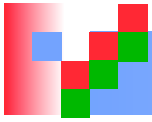
- May need multiple logical networks to guarantee forward progress with message passing



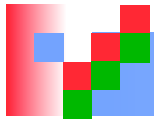
Which is better? SM or MP?



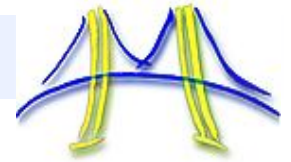
- Which is better, Shared Memory or Message Passing?
 - Depends on the program!
 - Both are "communication Turing complete"
 - » i.e. can build Shared Memory with Message Passing and vice-versa
- Advantages of Shared Memory:
 - Implicit communication (loads/stores)
 - Low overhead when cached
- Disadvantages of Shared Memory:
 - Complex to build in way that scales well
 - Requires synchronization operations
 - Hard to control data placement within caching system
- Advantages of Message Passing
 - Explicit Communication (sending/receiving of messages)
 - Easier to control data placement (no automatic caching)
- Disadvantages of Message Passing
 - Message passing overhead can be quite high
 - More complex to program
 - Introduces question of reception technique (interrupts/polling)



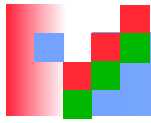
A Parallel Zoo Of Architectures



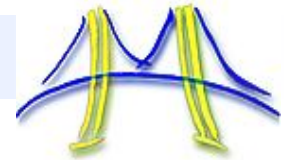
MIMD Machines



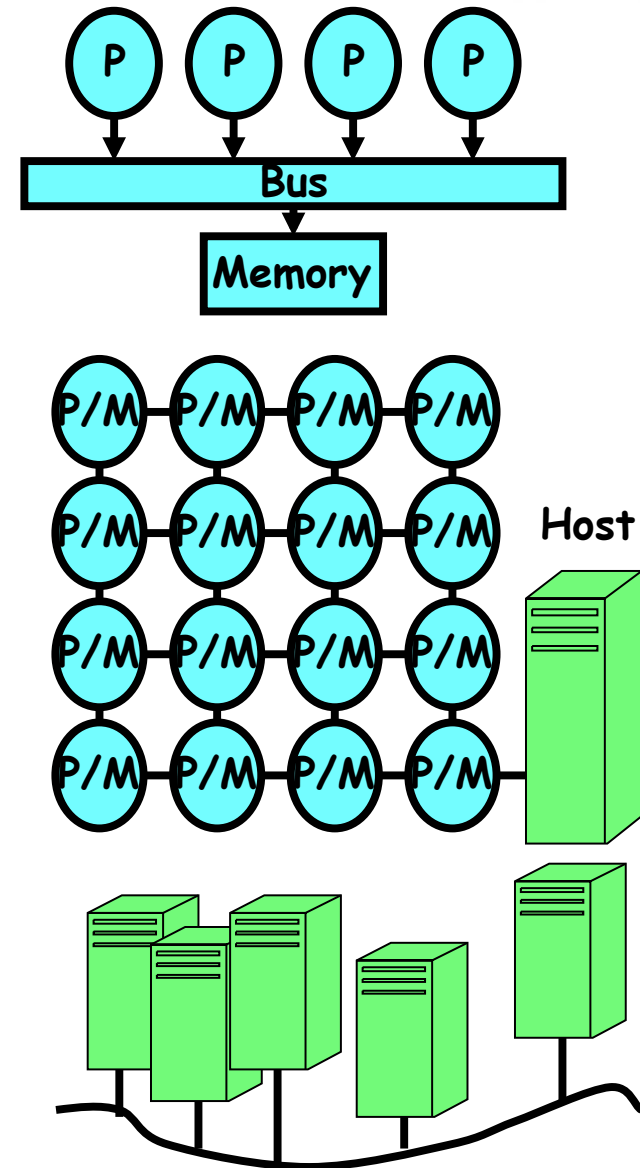
- **Multiple Instruction, Multiple Data (MIMD)**
 - Multiple independent instruction streams, program counters, etc
 - Called "multiprocessing" instead of "multithreading"
 - » Although, each of the multiple processors may be multithreaded
 - When independent instruction streams confined to single chip, becomes a "multicore" processor
- **Shared memory: Communication through Memory**
 - Option 1: no hardware global cache coherence
 - Option 2: hardware global cache coherence
- **Message passing: Communication through Messages**
 - Applications send explicit messages between nodes in order to communicate
- **For Most machines, Shared Memory built on top of message-passing network**
 - Bus-based machines are "exception"

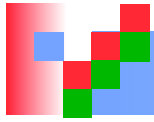


Examples of MIMD Machines



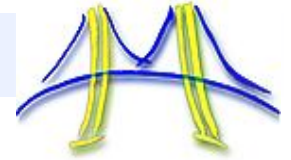
- Symmetric Multiprocessor
 - Multiple processors in box with shared memory communication
 - Current MultiCore chips like this
 - Every processor runs copy of OS
- Non-uniform shared-memory with separate I/O through host
 - Multiple processors
 - » Each with local memory
 - » general scalable network
 - Extremely light "OS" on node provides simple services
 - » Scheduling/synchronization
 - Network-accessible host for I/O
- Cluster
 - Many independent machine connected with general network
 - Communication through messages



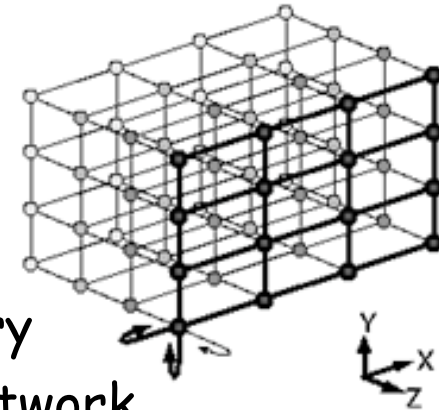


Cray T3E (1996)

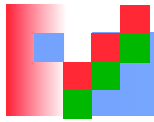
follow-on to earlier T3D (1993) using 21064's



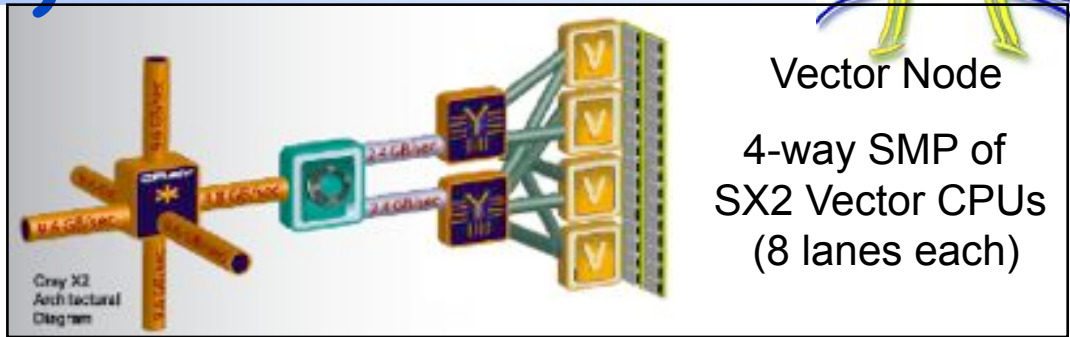
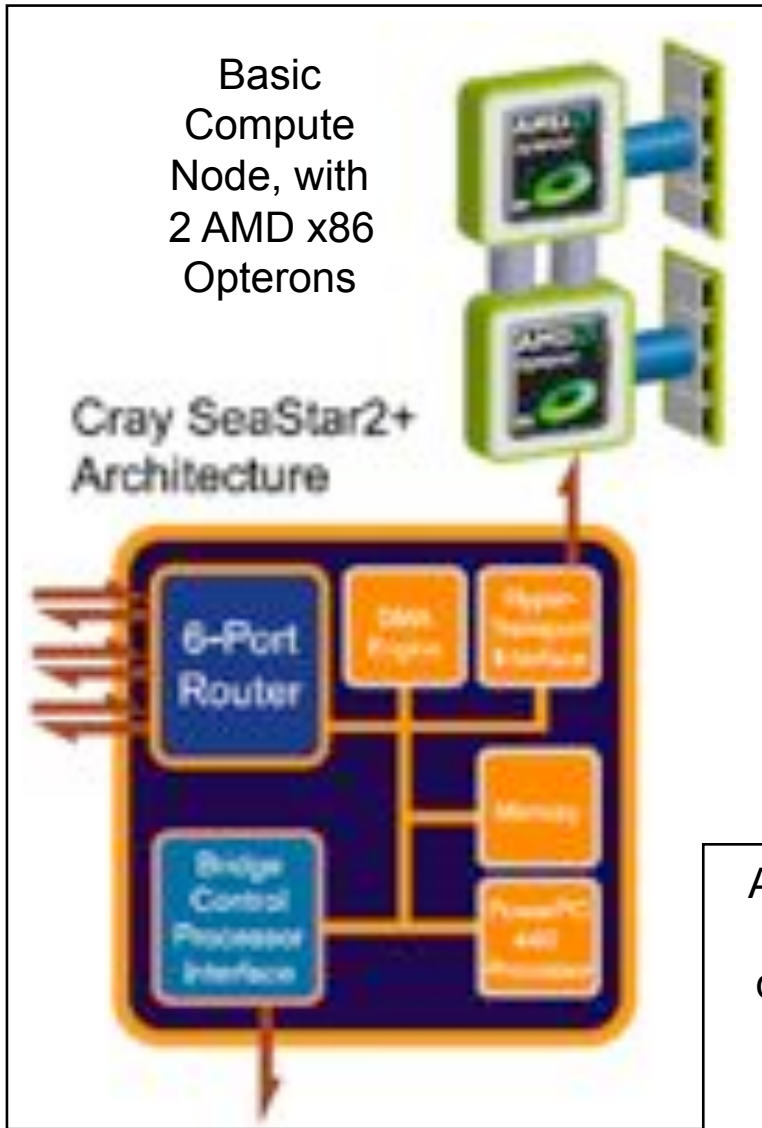
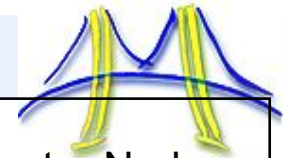
**Up to 2,048 675MHz Alpha 21164
processors connected in 3D torus**



- Each node has 256MB-2GB local DRAM memory
- Load and stores access global memory over network
- Only local memory cached by on-chip caches
- Alpha microprocessor surrounded by custom "shell" circuitry to make it into effective MPP node. Shell provides:
 - multiple stream buffers instead of board-level (L3) cache
 - external copy of on-chip cache tags to check against remote writes to local memory, generates on-chip invalidates on match
 - 512 external E registers (asynchronous vector load/store engine)
 - address management to allow all of external physical memory to be addressed
 - atomic memory operations (fetch&op)
 - support for hardware barriers/eureka to synchronize parallel tasks



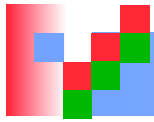
Cray XT5 (2007)



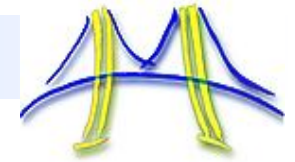
Also, XMT Multithreaded Nodes based on MTA design (128 threads per processor)

Processor plugs into Opteron socket

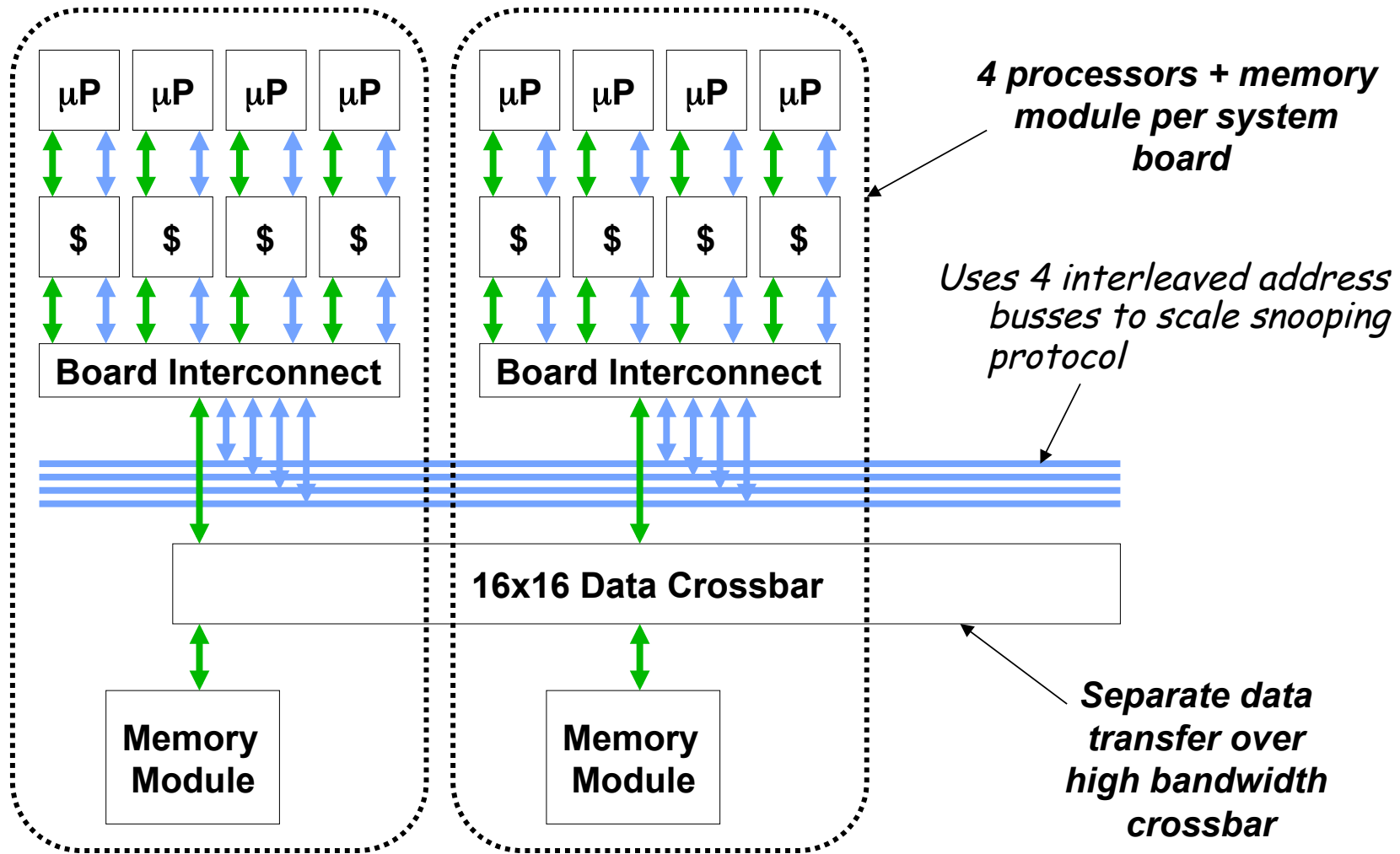




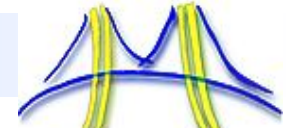
Sun Starfire UE10000 (1997)



Up to 64-way SMP using bus-based snooping protocol



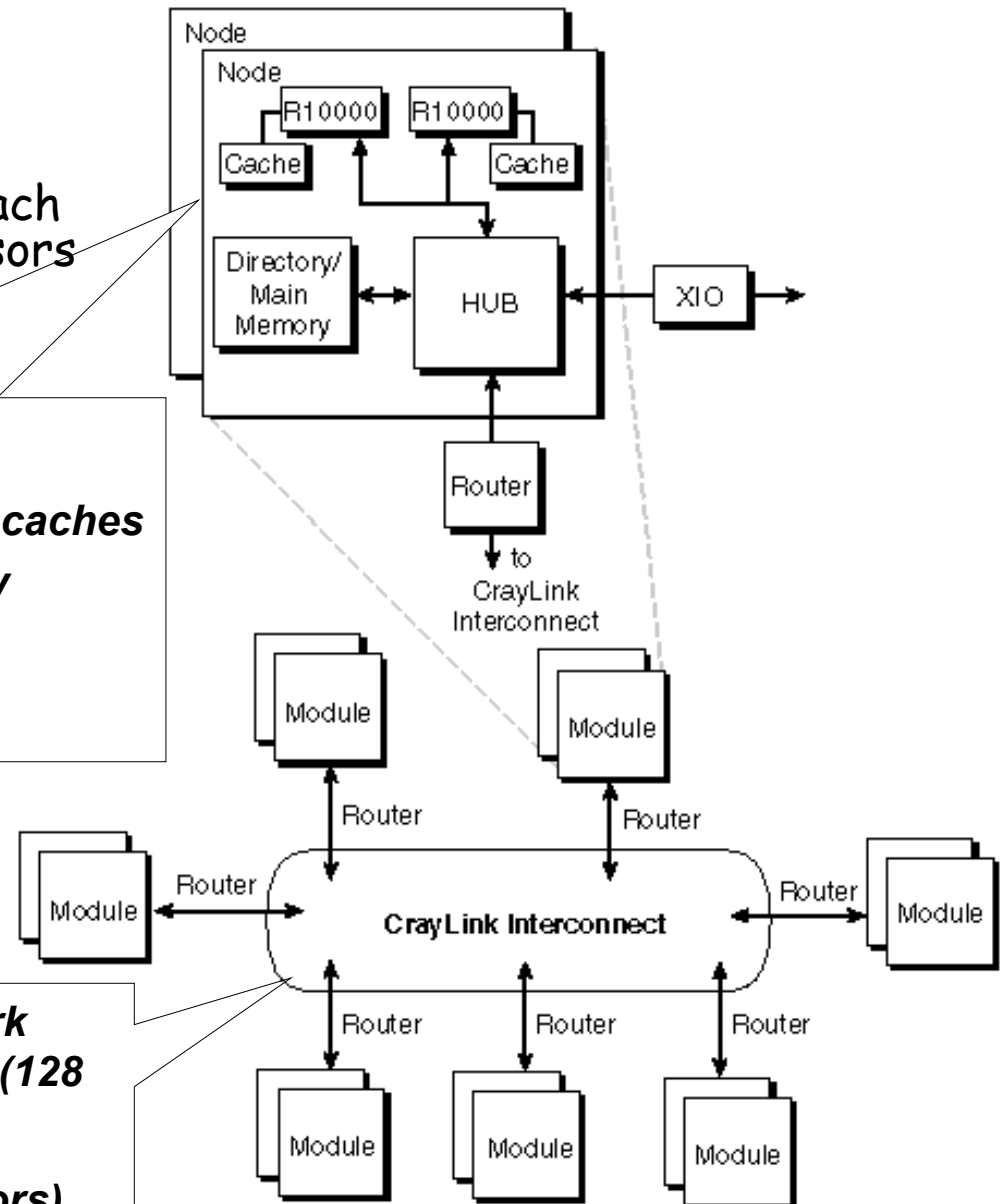
SGI Origin 2000 (1996)



- Large-Scale Distributed Directory SMP
 - Scales from 2 to 512 nodes
 - Direct-mapped directory with each bit standing for multiple processors
 - Not highly scalable beyond this

Node contains:

- Two MIPS R10000 processors plus caches
- Memory module including directory
- Connection to global network
- Connection to I/O

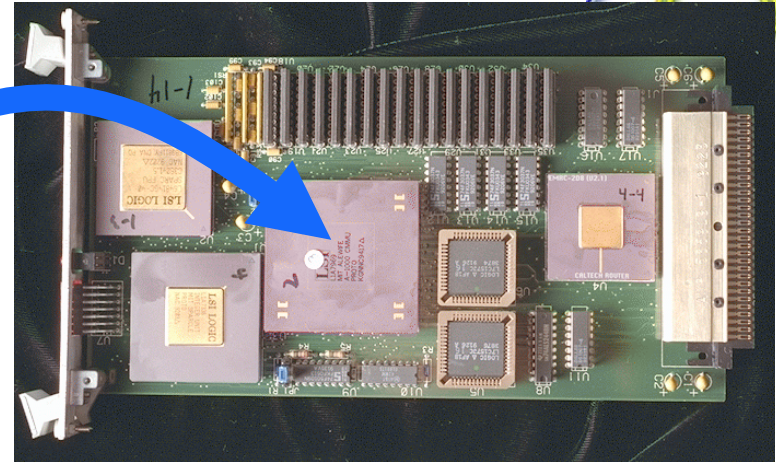
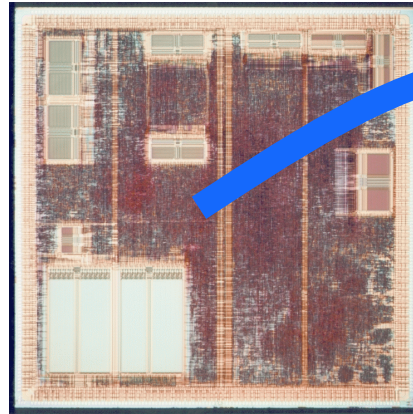
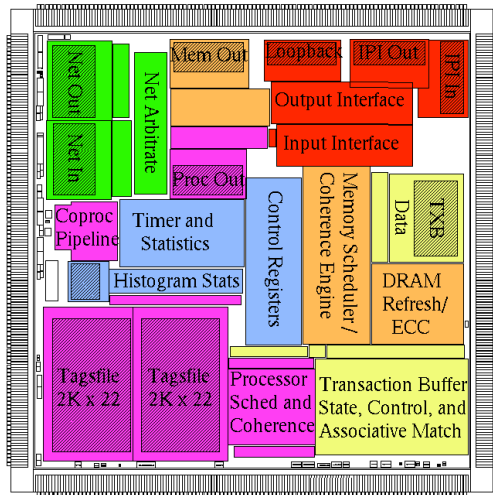


Scalable hypercube switching network supports up to 64 two-processor nodes (128 processors total)

(Some installations up to 512 processors)

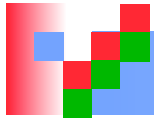
The Alewife Multiprocessor: SM & MP

Alewife-1000 CMMU



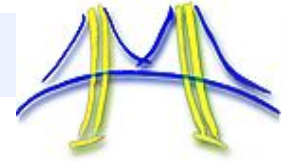
- Cache-coherence Shared Memory
 - Partially in Software!
 - Sequential Consistency
 - LimitLESS cache coherence for better scalability
- User-level Message-Passing
 - Fast, atomic launch of messages
 - Active messages
 - User-level interrupts
- Rapid Context-Switching
 - Course-grained multithreading
- Single Full/Empty bit per word for synchronization
 - Can build locks, barriers, other higher-level constructs





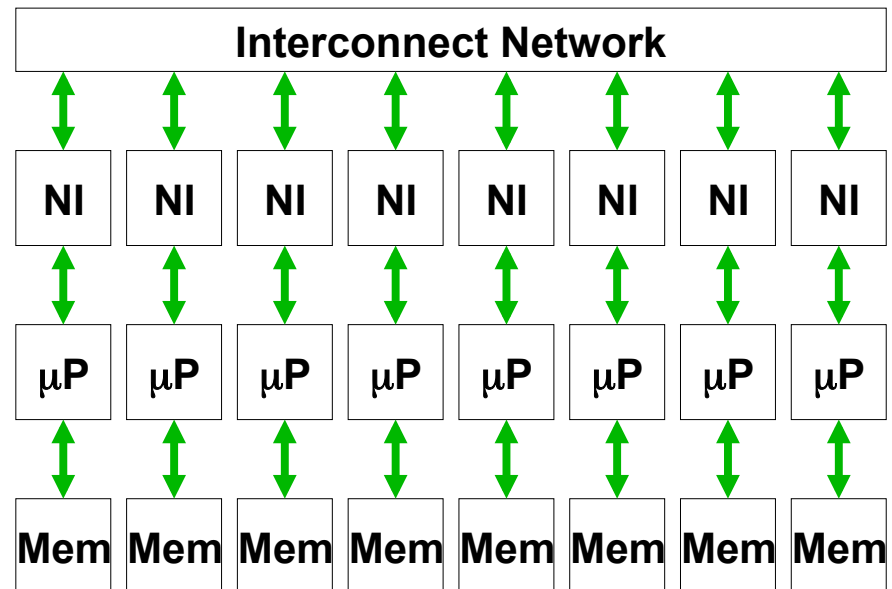
Message Passing MPPs

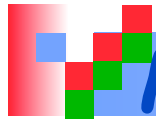
(Massively Parallel Processors)



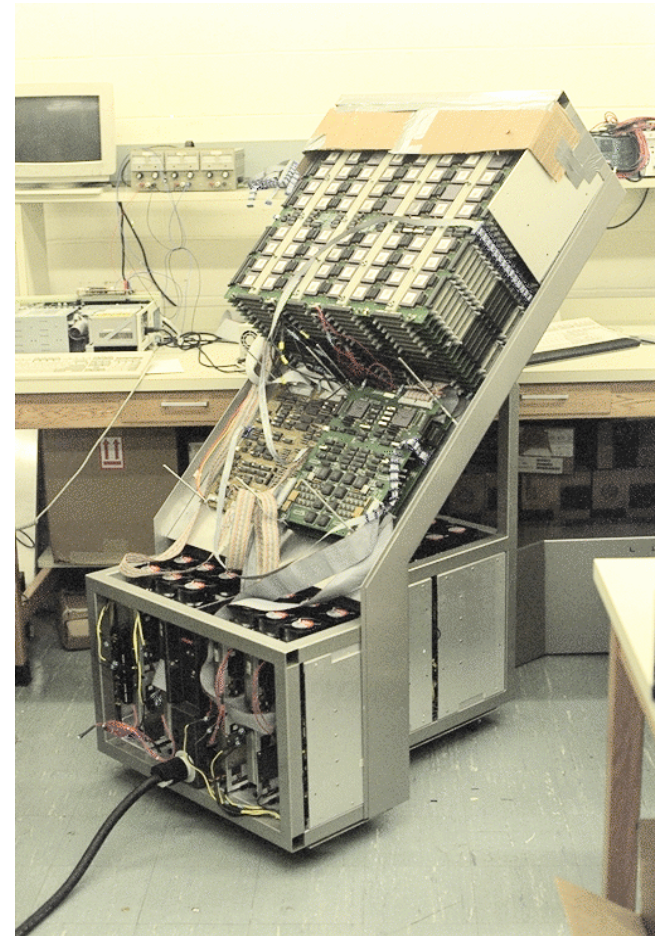
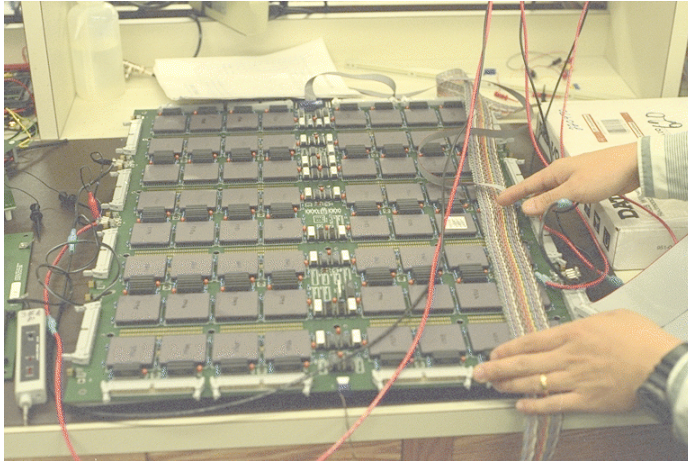
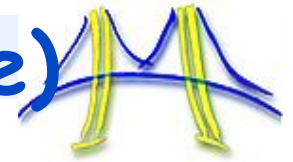
- Initial Research Projects
 - Caltech Cosmic Cube (early 1980s) using custom Mosaic processors
 - J-Machine (early 1990s) MIT
- Commercial Microprocessors including MPP Support
 - Transputer (1985)
 - nCube-1(1986) /nCube-2 (1990)
- Standard Microprocessors + Network Interfaces
 - Intel Paragon/i860 (1991)
 - TMC CM-5/SPARC (1992)
 - Meiko CS-2/SPARC (1993)
 - IBM SP-1/POWER (1993)
- MPP Vector Supers
 - Fujitsu VPP500 (1994)

**Designs scale to 100s-10,000s
of nodes**

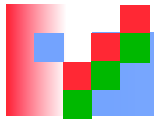




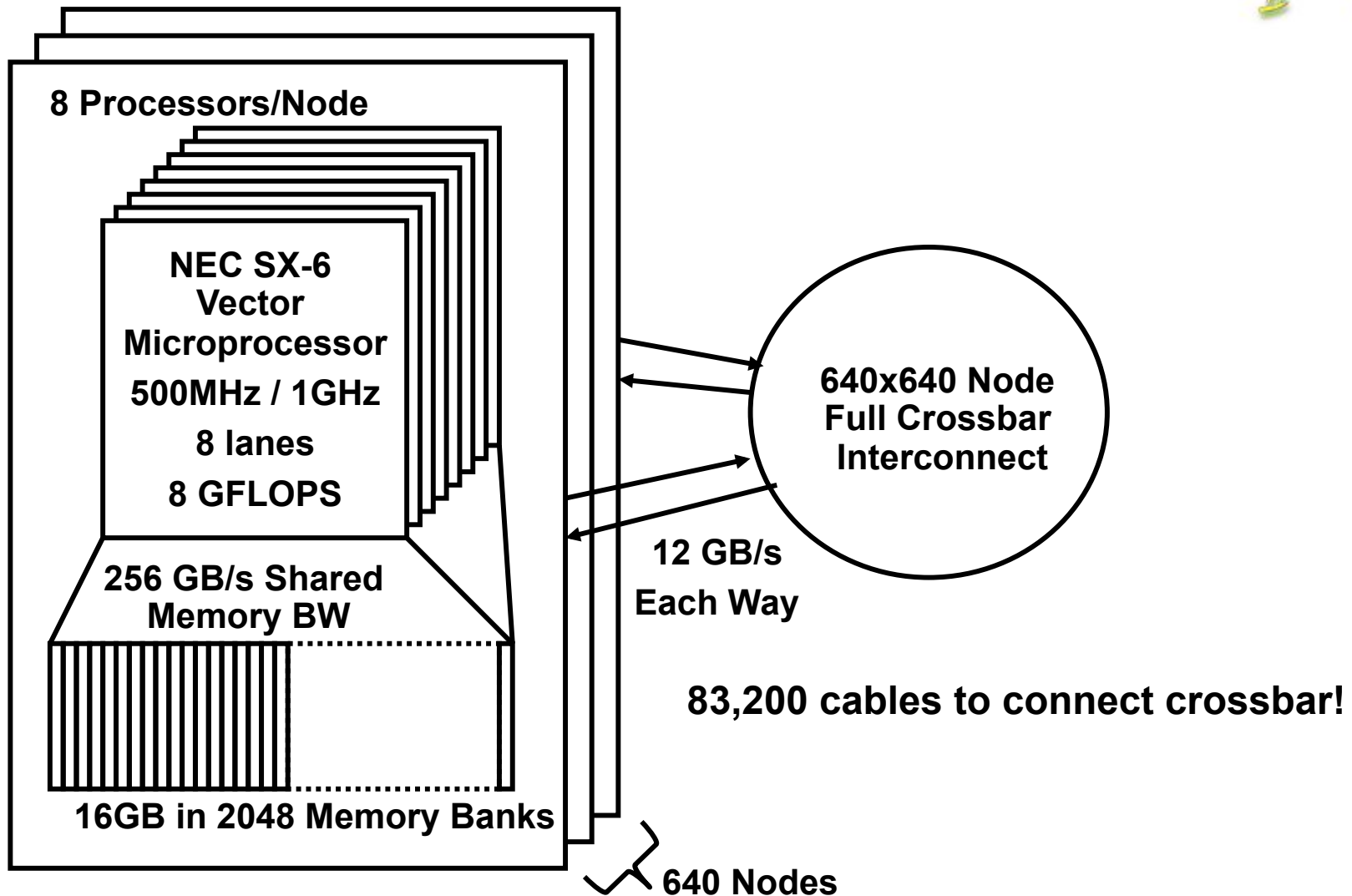
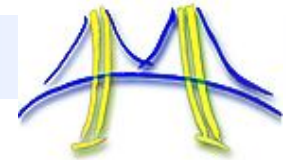
MIT J-Machine (Jelly-bean machine)



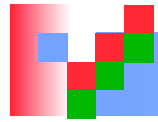
- 3-dimensional network topology
 - Non-adaptive, E-cubed routing
 - Hardware routing
 - Maximize density of communication
- 64-nodes/board, 1024 nodes total
- *Low-powered processors*
 - Message passing instructions
 - Associative array primitives to aid in synthesizing shared-address space
- *Extremely fine-grained communication*
 - Hardware-supported Active Messages



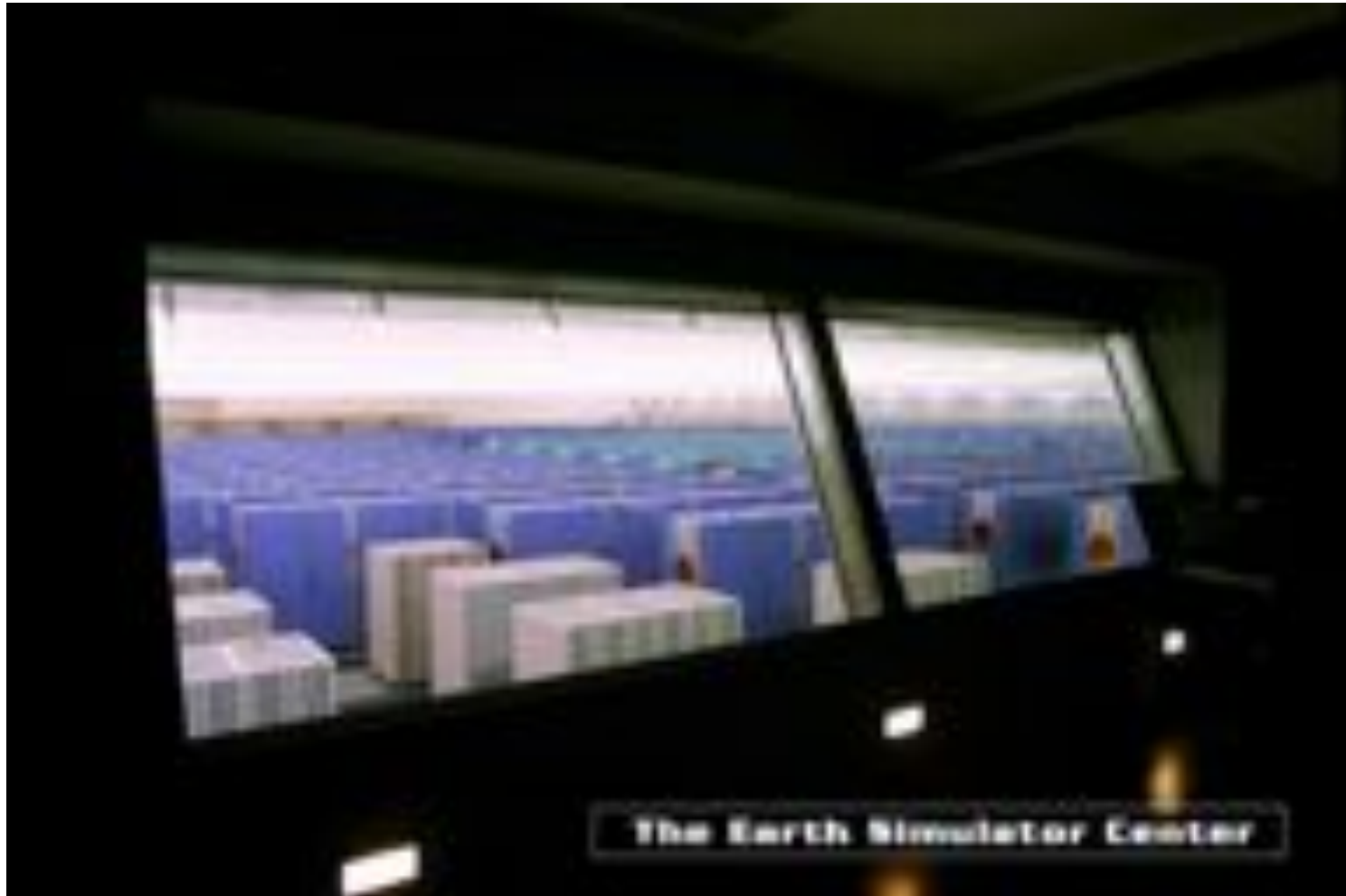
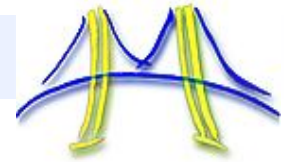
The Earth Simulator (NEC, 2002)

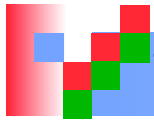


Was World's fastest supercomputer, >35 TFLOPS on LINPACK (June 2002)

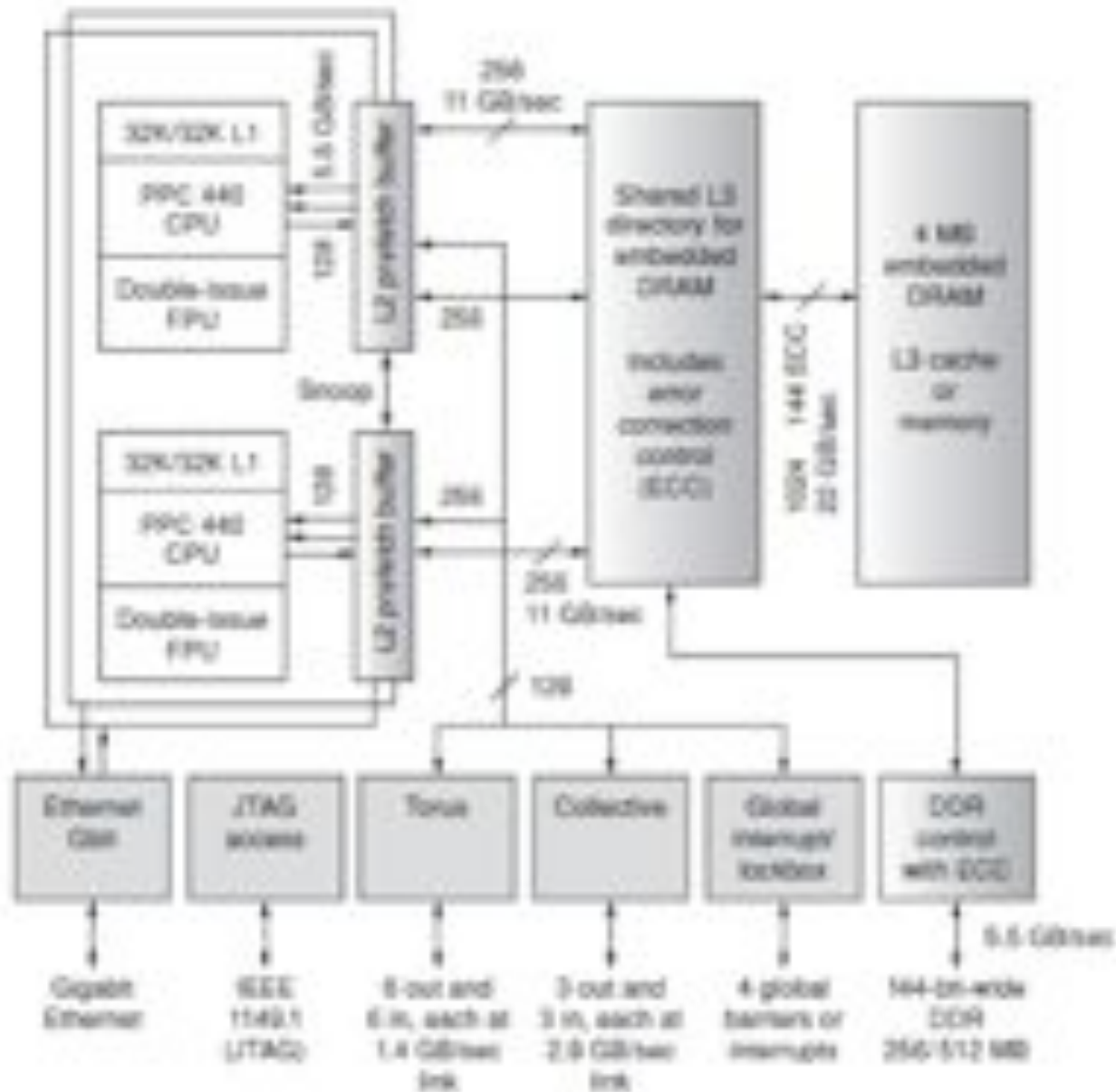
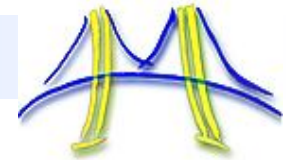


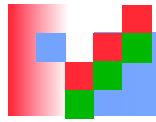
The Earth Simulator (NEC, 2002)



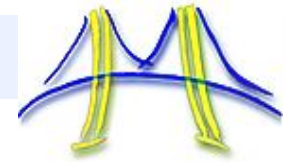


IBM Blue Gene/L Processor



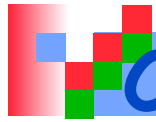


BG/L 64K Processor System



- Peak Performance 360TFLOPS
- Power Consumption 1.4 MW

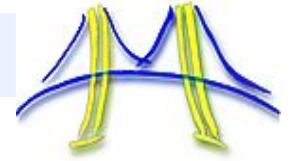
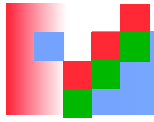




Clusters and Networks of Workstations

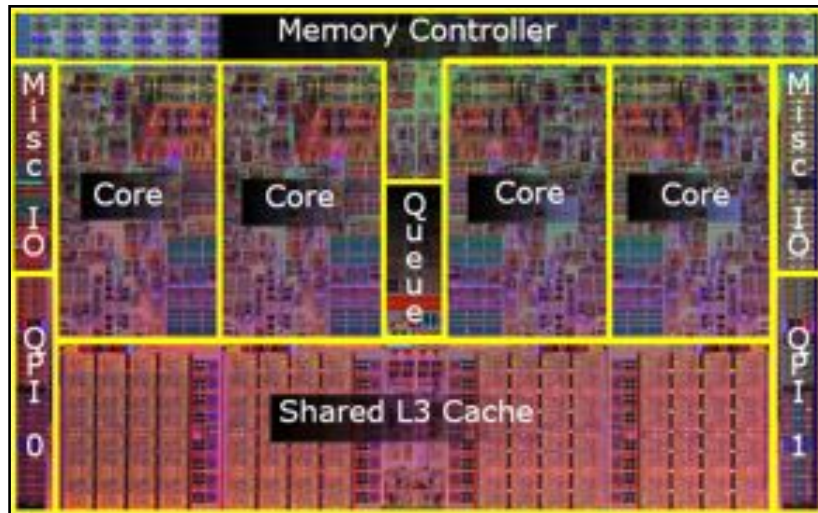


- Connect multiple complete machines together using standard fast interconnects
 - Little or no hardware development cost
 - Each node can boot separately and operate independently
 - Interconnect can be attached at I/O bus (most common) or on memory bus (higher speed but more difficult)
- Berkeley Project: "Network of Workstations" (NOW)
- Clustering initially used to provide fault tolerance
- Clusters of SMPs (CluMPs)
 - Connect multiple n-way SMPs using a cache-coherent memory bus, fast message passing network or non cache-coherent interconnect
- Build message passing MPP by connecting multiple workstations using fast interconnect connected to I/O Bus. Main advantage?

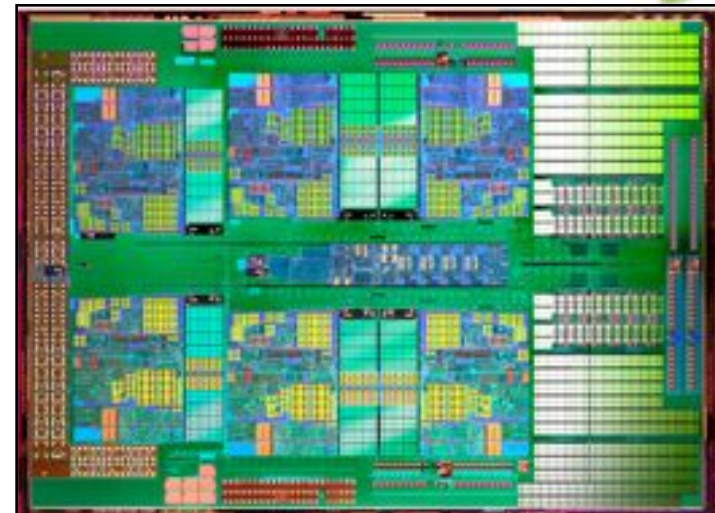


MultiCore Architectures

Parallel Chip-Scale Processors



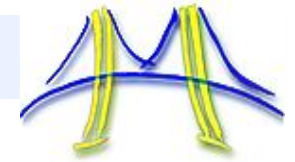
Intel Core 2 Quad: 4 Cores



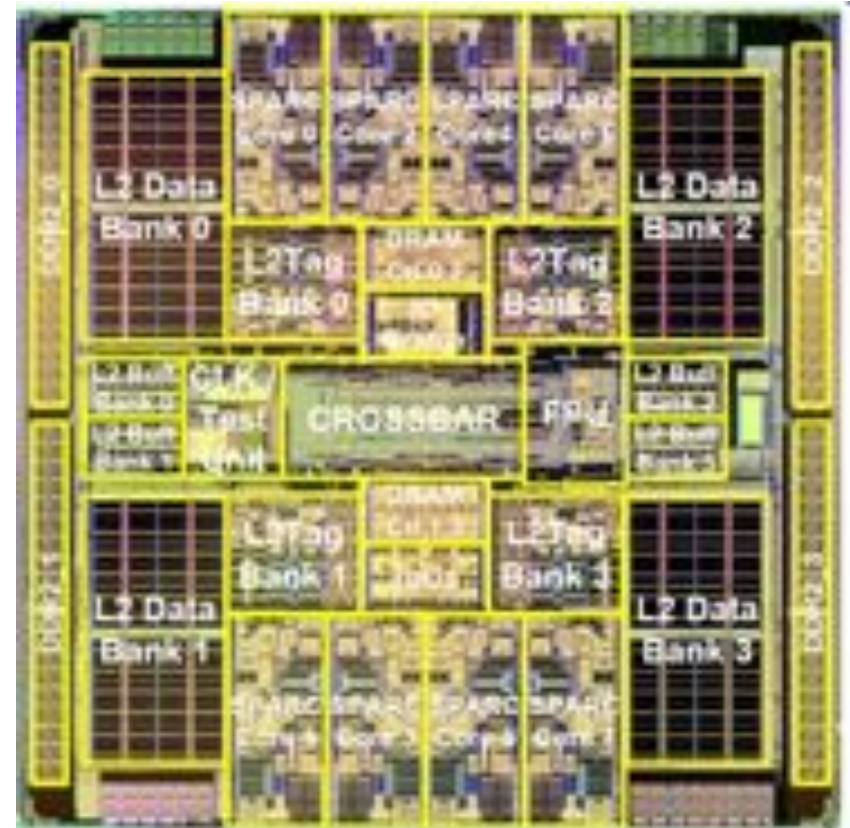
AMD Opteron: 6 Cores

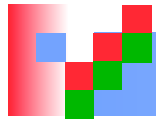
- Multicore processors emerging in general-purpose market due to power limitations in single-core performance scaling
 - 4-16 cores in 2009, connected as cache-coherent SMP
 - Cache-coherent shared memory
- Embedded applications need large amounts of computation
 - Recent trend to build "extreme" parallel processors with dozens to hundreds of parallel processing elements on one die
 - Often connected via on-chip networks, with no cache coherence
 - Examples: 188 core "Metro" chip from CISCO

T1 ("Niagara")

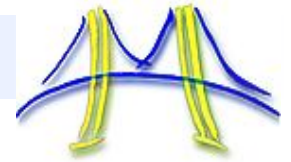


- Highly Threaded:
 - 8 Cores
 - 4 Threads/Core
- Target: Commercial server applications
 - High thread level parallelism (TLP)
 - » Large numbers of parallel client requests
 - Low instruction level parallelism (ILP)
 - » High cache miss rates
 - » Many unpredictable branches
 - » Frequent load-load dependencies
- Power, cooling, and space are major concerns for data centers
- Metric: Performance/Watt/Sq. Ft.
- Approach: Multicore, Fine-grain multithreading, Simple pipeline, Small L1 caches, Shared L2

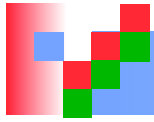




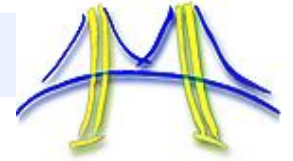
T1 Fine-Grained Multithreading



- Each core supports four threads and has its own level one caches (16KB for instructions and 8 KB for data)
 - Coherency is enforced among the L1 caches by a directory associated with each L2 cache block
- Switching to a new thread on each clock cycle
- Idle threads are bypassed in the scheduling
 - Waiting due to a pipeline delay or cache miss
 - Processor is idle only when all 4 threads are idle or stalled
- Both loads and branches incur a 3 cycle delay that can only be hidden by other threads
- A single set of floating-point functional units is shared by all 8 cores
 - floating-point performance was not a focus for T1
 - (New T2 design has FPU per core)

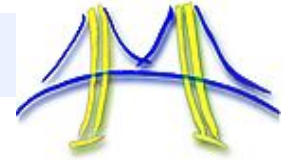


Embedded Parallel Processors

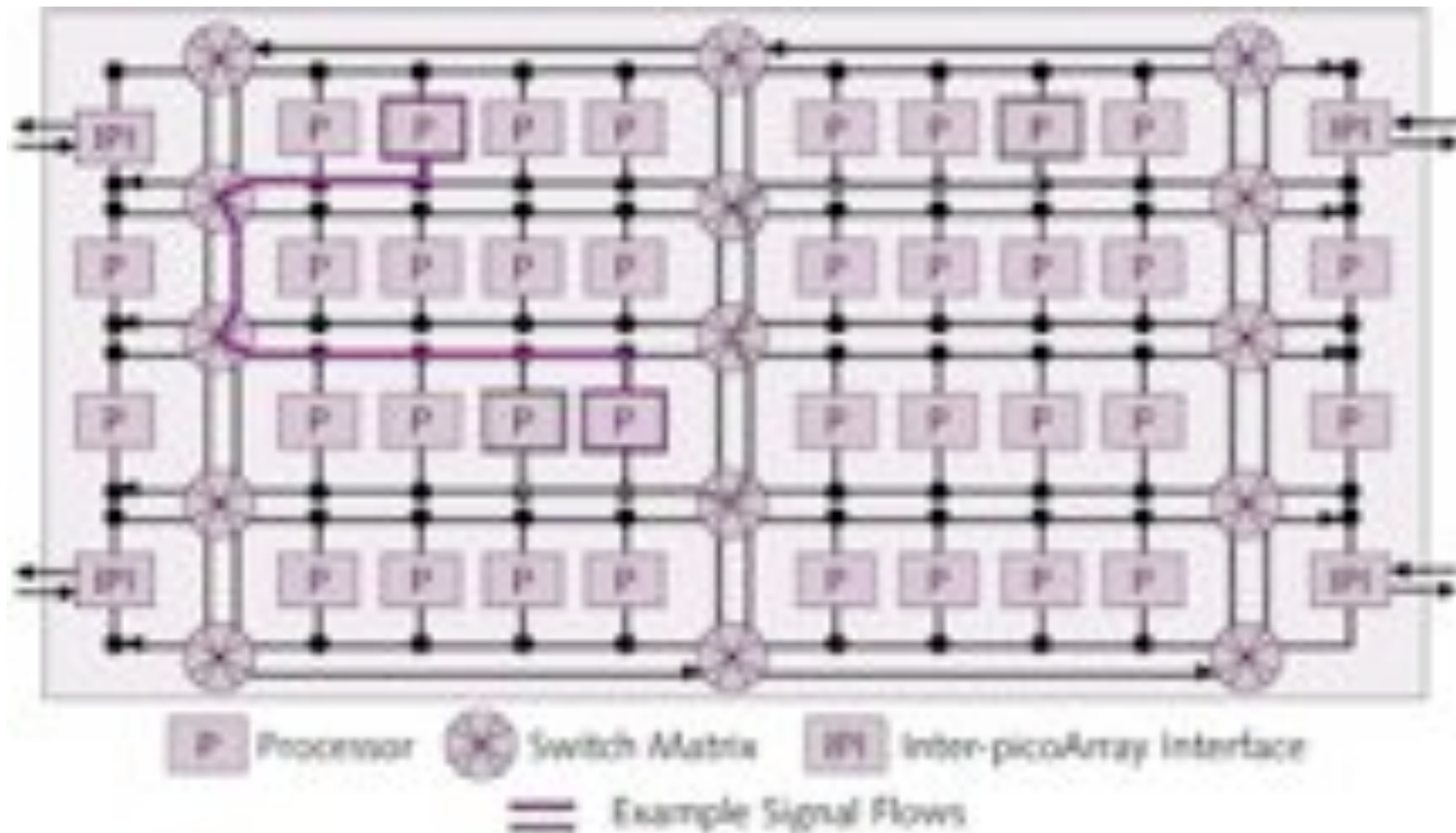


- Often embody a mixture of old architectural styles and ideas
- Exposed memory hierarchies and interconnection networks
 - Programmers code to the "metal" to get best cost/power/performance
 - Portability across platforms less important
- Customized synchronization mechanisms
 - Interlocked communication channels (processor blocks on read if data not ready)
 - Barrier signals
 - Specialized atomic operation units
- Many more, simpler cores

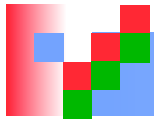
PicoChip PC101 (2003)



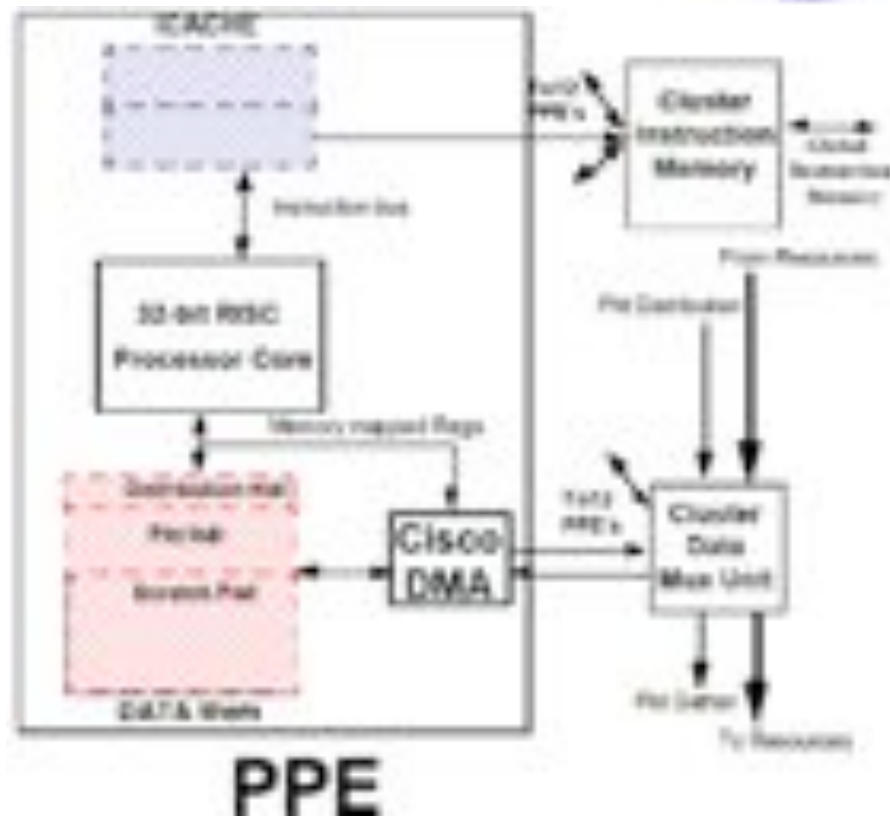
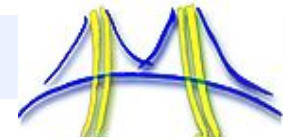
- Target market is wireless base stations
- 430 cores on one die in 130nm
- Each core is a 3-issue VLIW



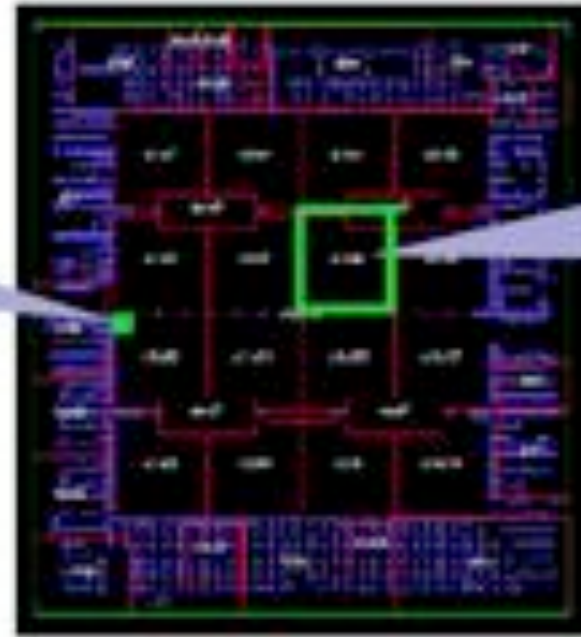
[uPR, July 2003]



Cisco CSR-1 Metro Chip

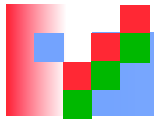


.5sqmm per PPE

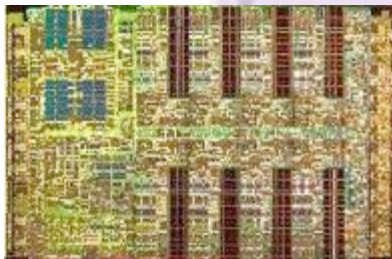
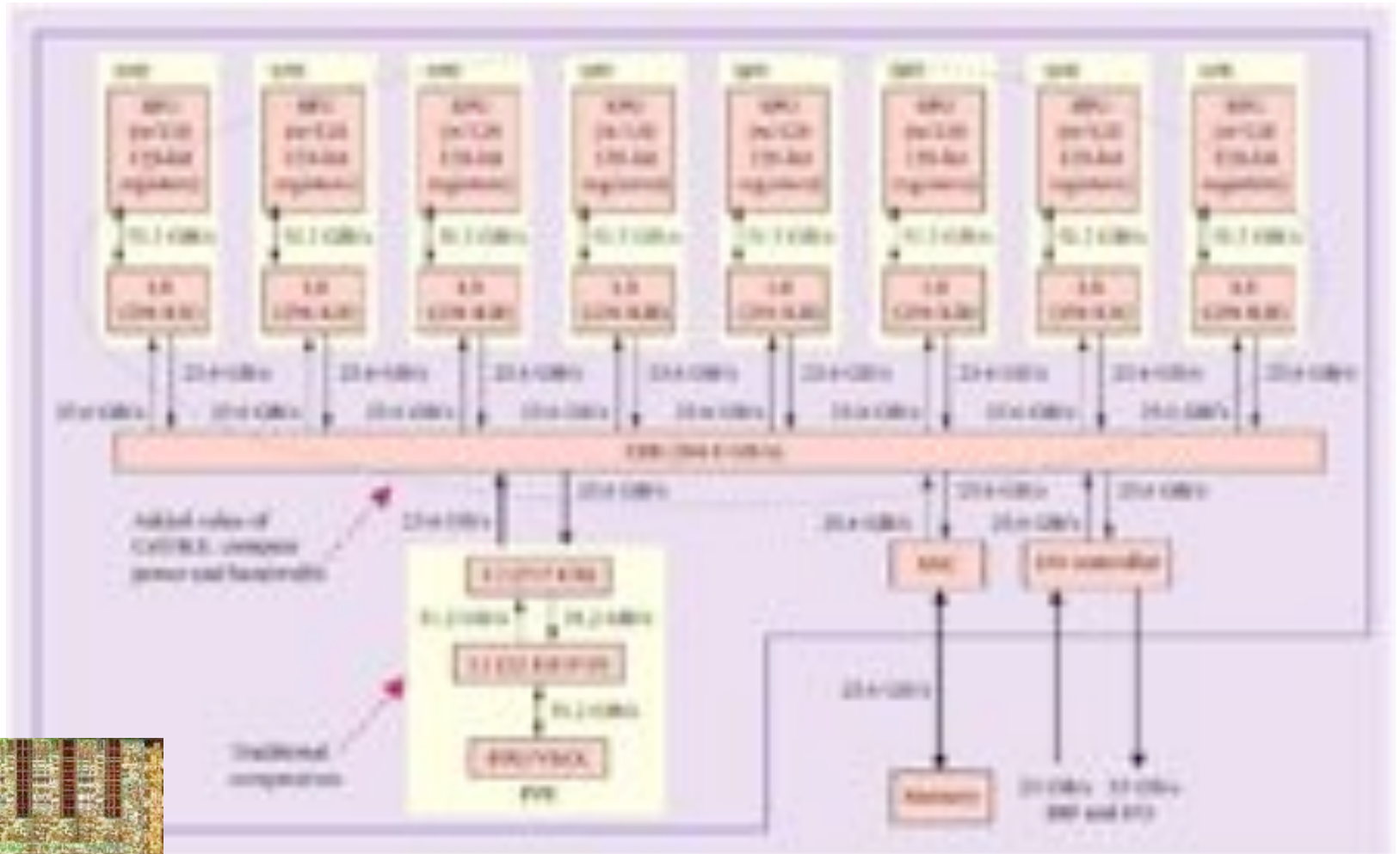
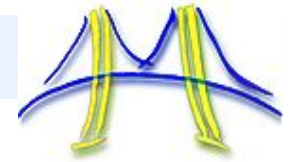


16 PPE Clusters
Each Cluster of 12 PPE's

188 usable RISC-like cores in 130nm

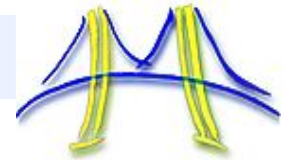


IBM Cell Processor (Playstation-3)

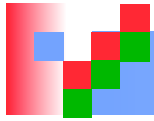


One 2-way threaded PowerPC core (PPE), plus eight specialized short-SIMD cores (SPE)

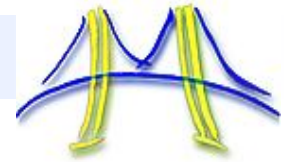
Nvidia G8800 Graphics Processor



- This is a GPU (Graphics Processor Unit)
 - Available in many desktops
- Example: 16 cores similar to a vector processor with 8 lanes (128 stream processors total)
 - Processes threads in SIMD groups of 32 (a "warp")
 - Some stripmining done in hardware
- Threads can branch, but loses performance compared to when all threads are running same code
- Complete parallel programming environment (CUDA)
 - A lot of parallel codes have been ported to these GPUs
 - For some data parallel applications, GPUs provide the fastest implementations



Conclusion



- **Uniprocessor Parallelism:**
 - Pipelining, Superscalar, Out-of-order execution
 - Vector Processing, Pseudo-SIMD
- **Multithreading**
 - Multiple independent threads executing on same processor
- **Memory Systems:**
 - Exploiting of Locality at many levels
 - Greatly Impacts performance (sometimes in strange fashion)
 - Use of Autotuning to navigate complexity
- **Shared Memory Programming Model:**
 - Multiple threads communicating through memory
 - Memory Consistency Model: Specifies order with which operations seem to occur relative to one another
 - Sequential Consistency: Most "intuitive" model
- **Message Passing Programming Model:**
 - Multiple threads communicating with messages