

Multiprocessors and Multiprocessing

more is better?

Parallelism in just one processor

A slow sort of country! said the Queen. *'Now, HERE, you see, it takes all the running YOU can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!'* from *Through the Looking Glass* by Lewis Carroll

Super-pipelining

Superscalar

Dynamic scheduling

Super-pipelining

The more the stages (and the smaller the stages) the faster the theoretical peak speed of the pipeline at least theoretically
😊

Some modern processors have 8 or more pipeline stages
See websurfing homework

Superscalar

What if we can issue more than one instruction per cycle?

What if we *replicate* our MIPS hardware

Imagine we have an integer ALU or branch pipeline and a load/store pipeline

ALU or Branch	IF ID EX MEM WB
Load or Store	IF ID EX MEM WB
ALU or Branch	IF ID EX MEM WB
Load or Store	IF ID EX MEM WB

Dynamic Scheduling

Keep a buffer of instructions and allow them to execute out of order to boost efficiency as long as semantics are preserved

Works well with superscalar

Superscalar

Hardware keeps track and issues only one instruction if there is a hazard

Read and decode two instructions (64 bits) per cycle

If an instruction cannot be issued right away buffer it

Extra read and write ports on register file

Superscalar

How would this loop be scheduled on superscalar MIPS?

```
Loop:   lw      $t0, 0($s1)
        addu   $t0,$t0,$s2
        sw     $t0, 0($s1)
        addi   $s1,$s1, -4
        bne   $s1, $zero, Loop
```

Superscalar

```
Loop:      lw      $t0, 0(#s1)
           addu   $t0,$t0,$s2
           sw     $t0, 0($s1)
           addi  $s1,$s1, -4
           bne   $s1, $zero, loop
```

<u>ALU or Branch</u>	<u>Load/Store</u>
	lw \$t0, 0(\$s1)
addi \$s1, \$s1, -4	
addu \$t0,\$t0,\$s2	
bne \$s1, \$zero, Loop	sw \$t0, 0(\$s1)

Superscalar

Previous was not too impressive

We need some help from our hereditary enemies in compilers

Loop unrolling

multiple copies of loop body are made and instructions from different iterations are scheduled together

Superscalar with loop unrolling

ALU or Branch

Load/Store

addi \$s1,\$s1,-16

lw \$t0, 0(\$s1)

lw \$t1, 12(\$s1)

addu \$t0,\$t0, \$s2

lw \$t2, 8(\$s1)

addu \$t1,\$t1, \$s2

lw \$t3, 12(\$s1)

addu \$t2,\$t2, \$s2

sw \$t0, 0(\$s1)

addu \$t3,\$t3, \$s2

sw \$t1, 12(\$s1)

sw \$t2, 8(\$s1)

bne \$s1, \$zero, Loop

sw \$t3, 4(\$s1)

Multiprocessors

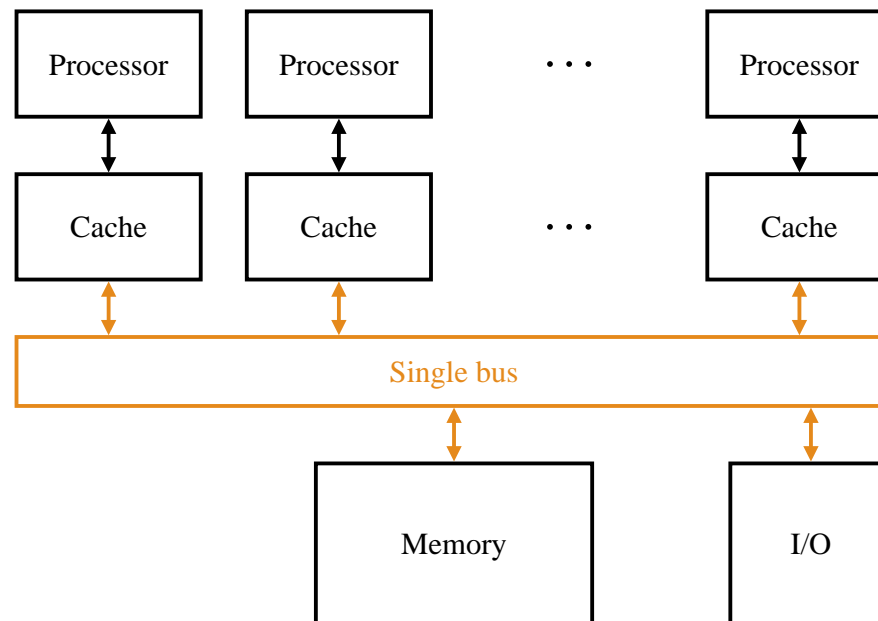
why would you want a multiprocessor?

what things can it do well?

What things can't it do well?

What things can it do that a *bunch of computers* can't do?

How much are you willing to pay?



Classifying Multiprocessors

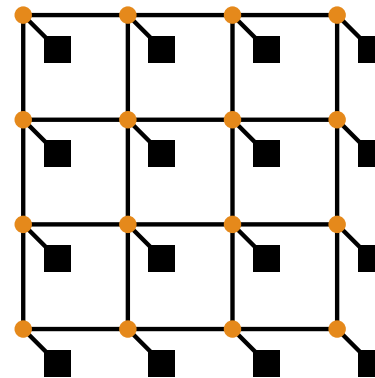
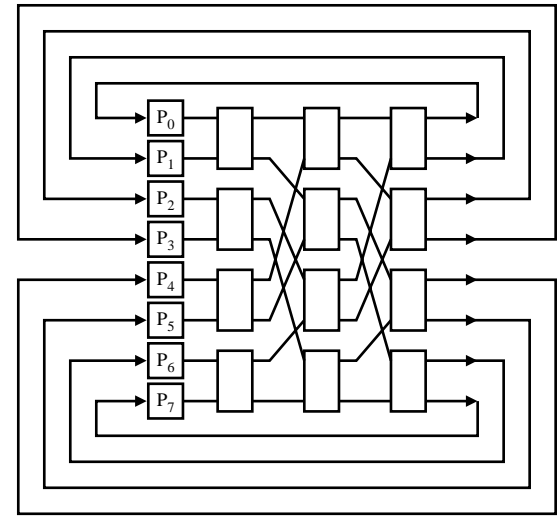
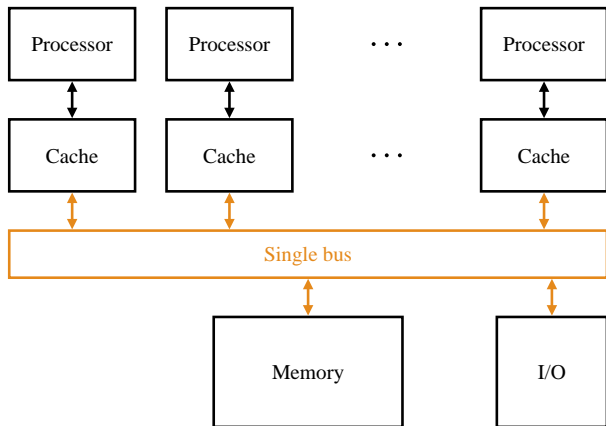
Interconnection Network

Memory Topology

Programming Model

Interconnection Network

Bus
Network
pros/cons?

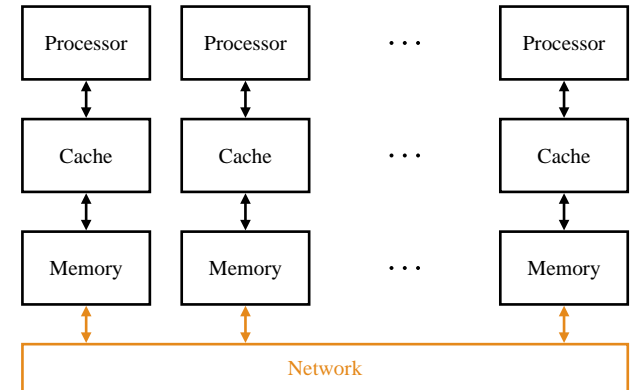
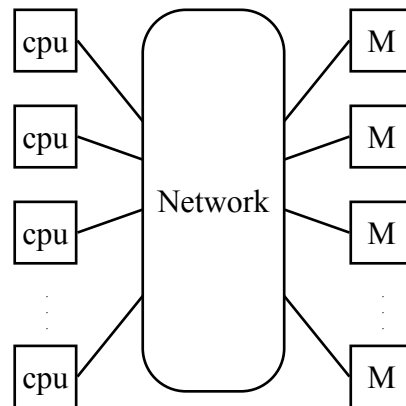
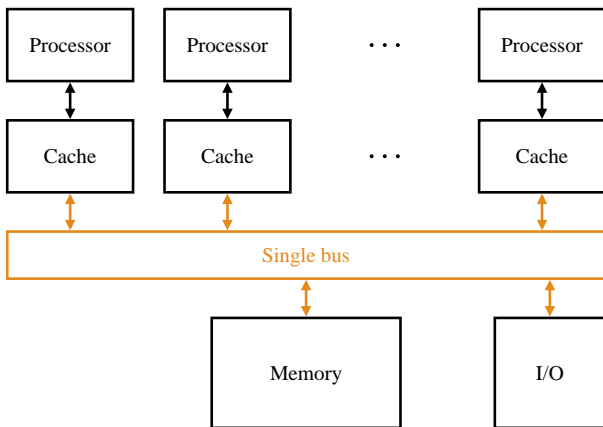
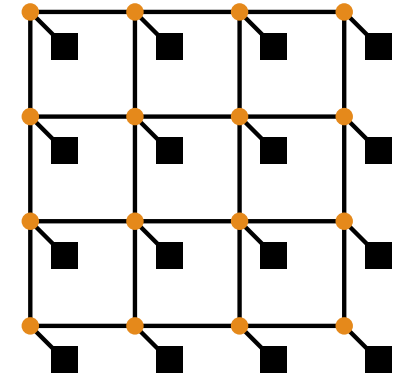


Memory Topology

UMA (Uniform Memory Access)

NUMA (Non-uniform Memory Access)

pros/cons?



Programming Model

Shared Memory -- every processor can name every address location

Message Passing -- each processor can name only it's local memory.

Communication is through explicit messages.

pros/cons?

find the max of 100,000 integers on 10 processors.

Parallel Programming

Processor A

index = i++;

i = 47

Processor B

index = i++;

Shared-memory programming requires synchronization to provide mutual exclusion and prevent race conditions

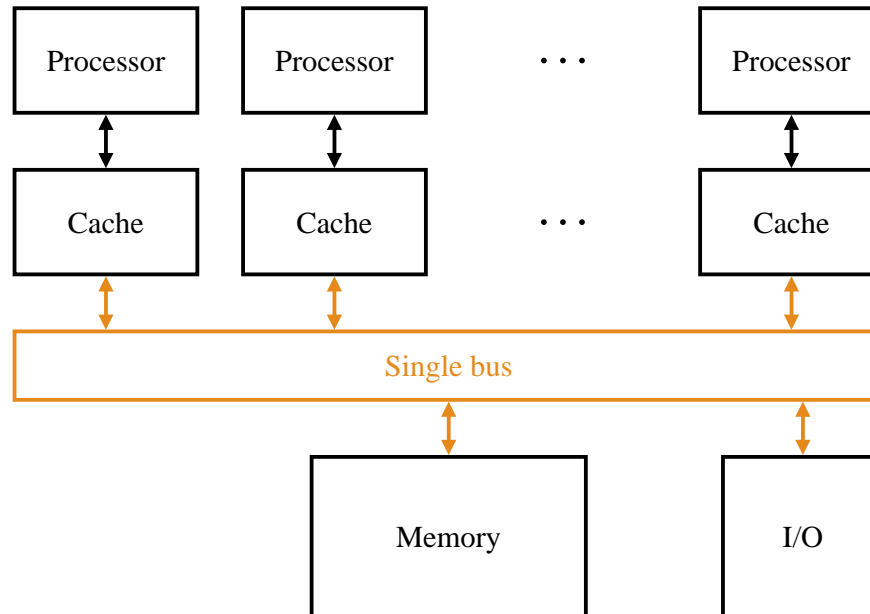
locks (semaphores)

barriers

Multiprocessor Caches (Shared Memory)

the problem -- cache coherency

the solution?



Cache Coherency

write-update

on each write, each cache holding that location updates its value

write-invalidate \leq most common

on each write, each cache holding that location invalidates the cache line.

both schemes MUCH easier on a bus-based multiprocessor
potentially requires a LOT of messages, but...

Cache Coherency

A good cache coherency protocol can avoid sending unnecessary (and expensive) invalidate or update messages.

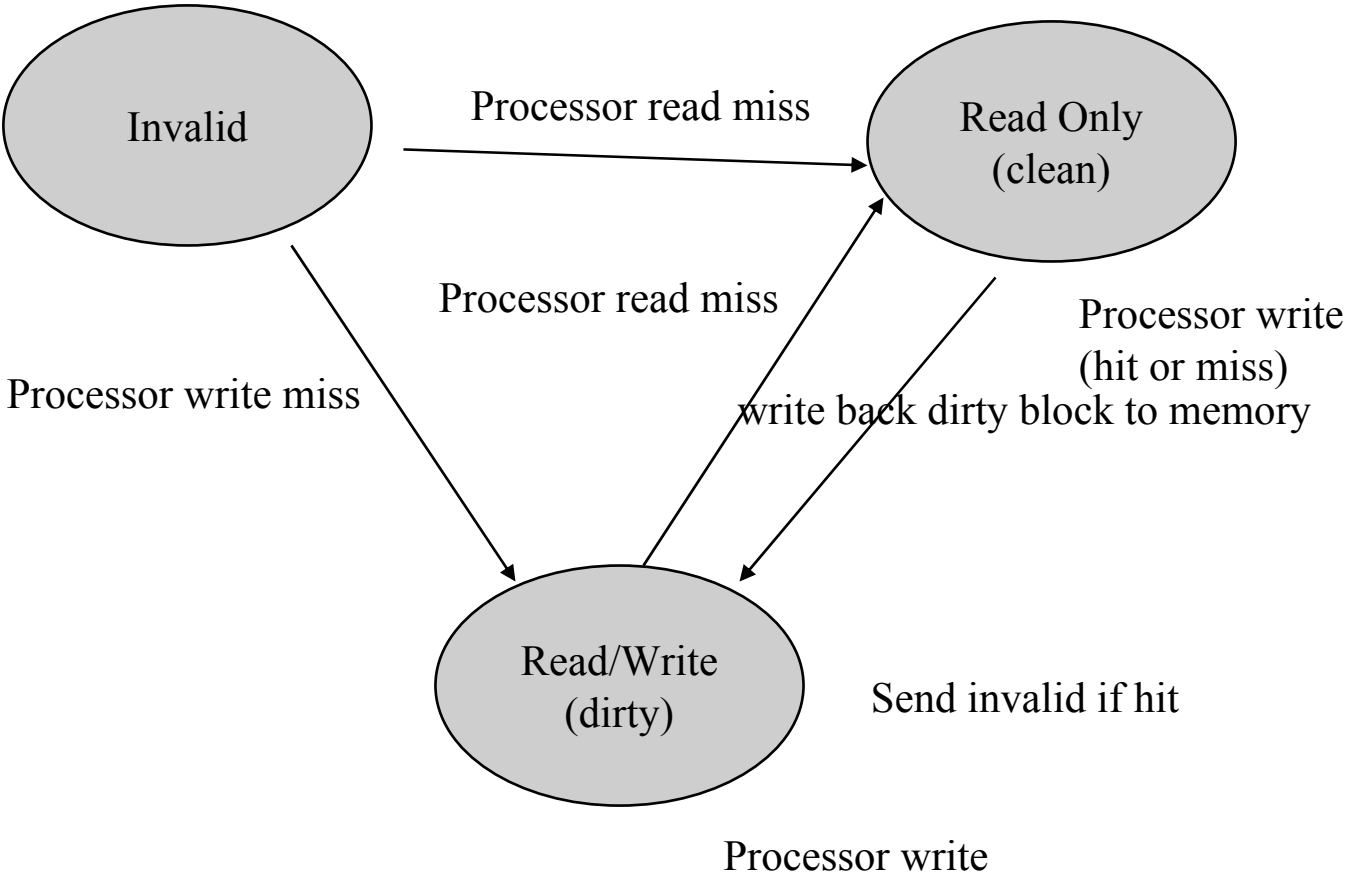
Allows each cache line to be in one of several *states*
e.g. for write-invalidation with write-back cache:

Read only: cache block is clean and may be shared

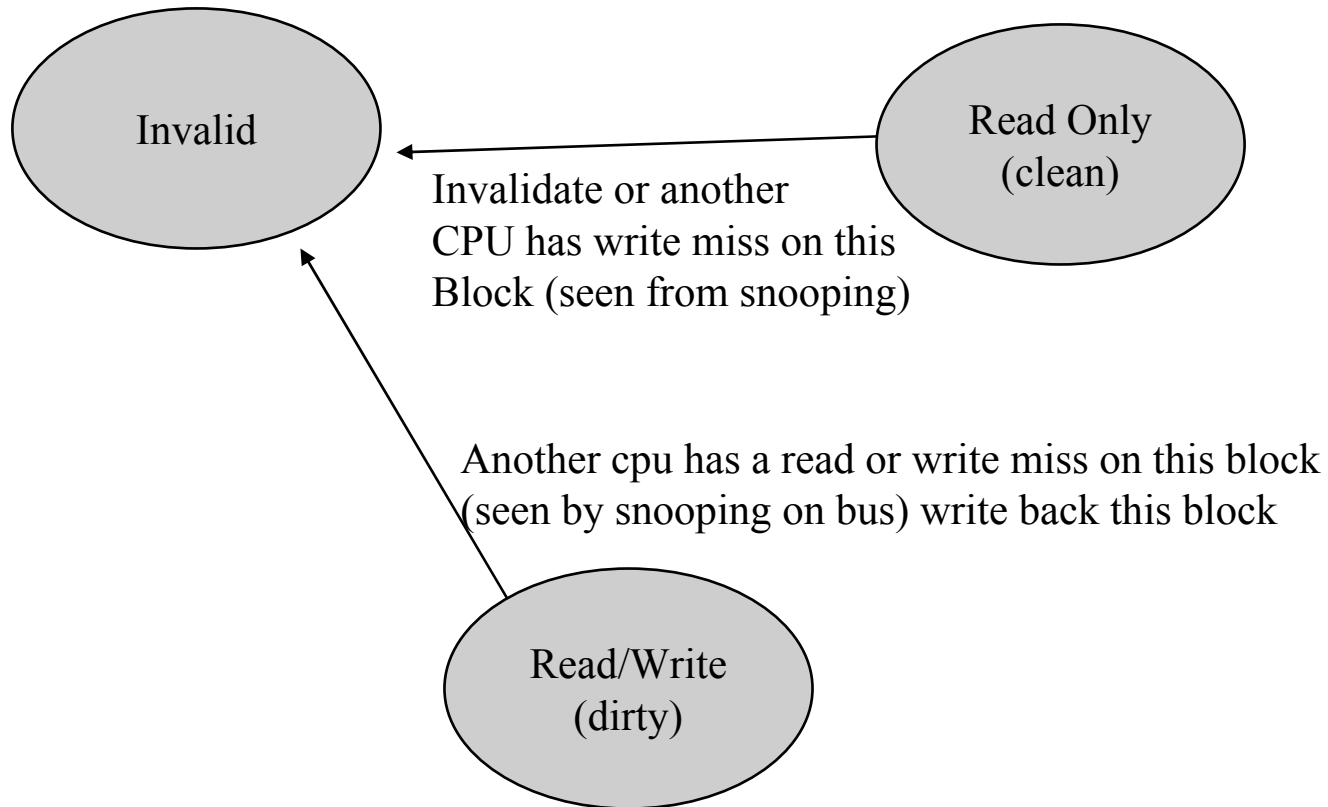
Read/Write: cache block is dirty and may not be shared

Invalid: data in block is invalid

Cache state using signals from processor



Cache state transitions on signals from bus

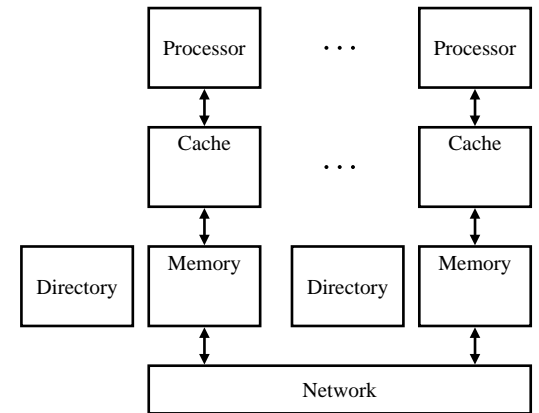
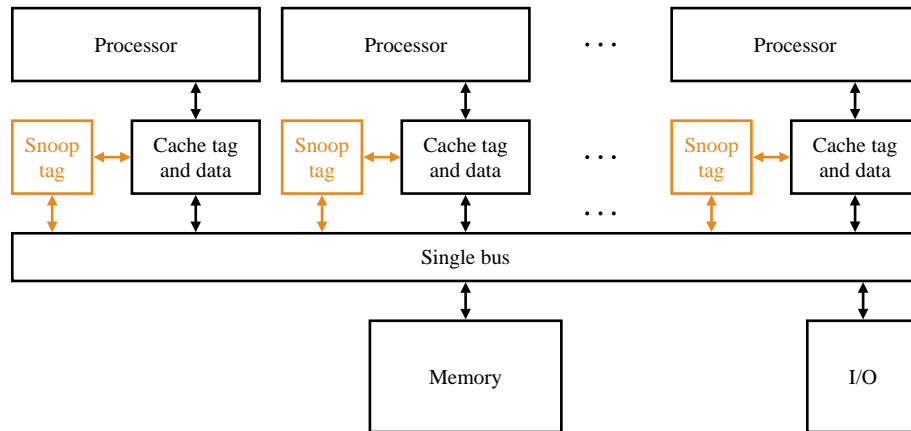


Cache Coherency

How do you know when an external read/write occurs?

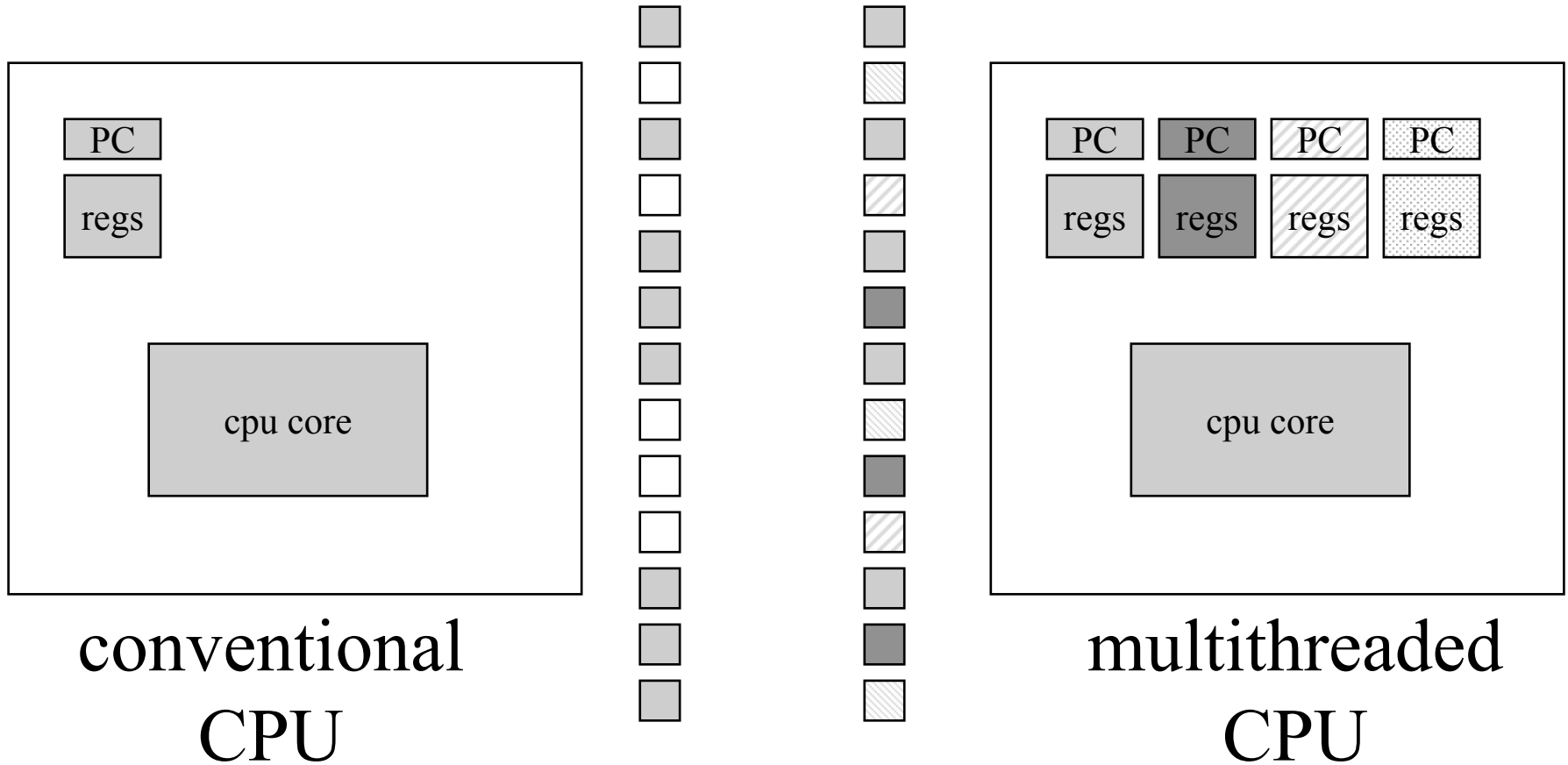
Snooping protocols

Directory protocols



Multithreading Processors

uniprocessor? multiprocessor?



conventional
CPU

multithreaded
CPU

Multithreaded Processors

Coarse-grain multithreading (Alewife-MIT)

context switch at long-latency operations (cache misses)

Fine-grain multithreading (Tera Supercomputer)

context switch every cycle

Simultaneous multithreading (Tullsen, Eggers, Levy 1995)

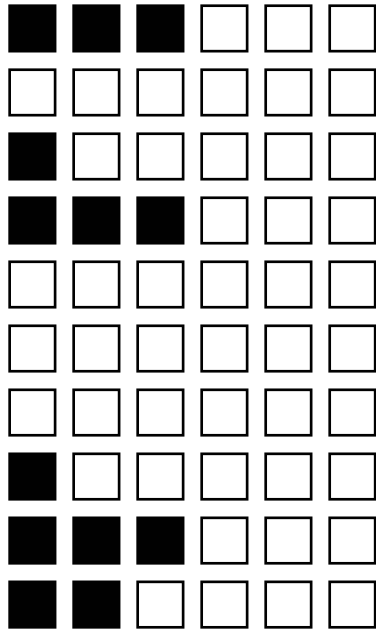
execute instructions from multiple threads in the same cycle

is only different from fine-grain multithreading in the context of
superscalar execution

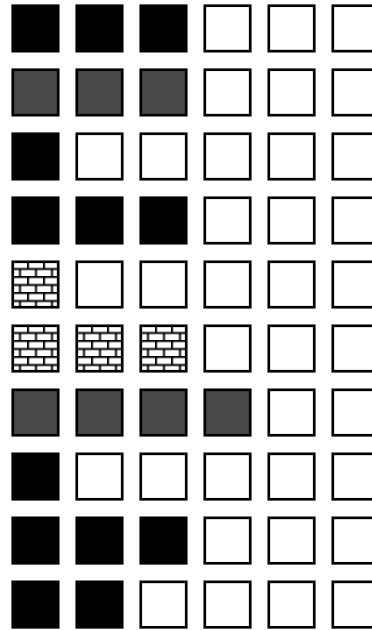
requires surprisingly few changes to a conventional out-of-order
superscalar processor

Intel calls it Hyperthreading

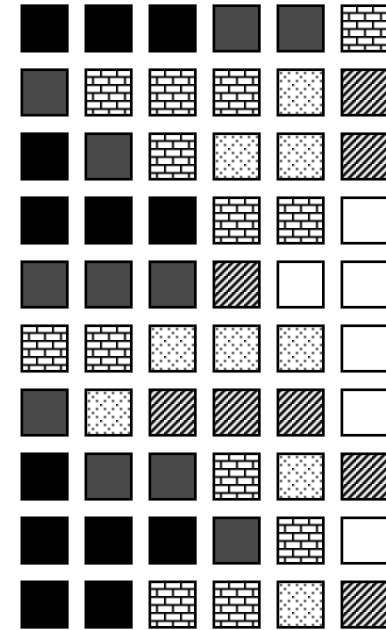
Multithreading models



Conventional
Superscalar



Fine-Grain
Multithreading



Simultaneous
Multithreading

Multiprocessors -- Key Points

Network vs. Bus

Message-passing vs. Shared Memory

Shared Memory is more intuitive, but creates problems for both the programmer (memory consistency, requiring synchronization) and the architect (cache coherency).

Multithreading gives the illusion of multiprocessing (including, in many cases, the performance) with very little additional hardware.