Parallelism

Parallel Systems perform multiple actions at the same time

Multitasking systems execute programs in parallel

- Example: surfing the web while playing music files
- or editing a document while updating a spreadsheet

Switching frequently between sequentially executing tasks

- Gives the illusion of continuous execution
- Tasks execute concurrently rather than simultaneously
- The tasks may run on the same processor

Forms of Parallelism

Pipelining overlaps multiple instructions
- Example of fine grained instruction level parallelism (ILP)
- Overlaps instruction steps (fetch, decode, execute, etc.)
- Scalar pipeline contains just one execution unit

Superscalar processors allow true parallel processing
- Takes place within the CPU
- Executes separate instructions simultaneously
- Invisible to the user
- Compiler assists in supplying more instructions
- Separate execute units may each be pipelined

Forms of Parallelism

Hyperthreading is another form of parallelism
- Single execute unit switches between threads
- Registers & PC are replicated (one set per thread)
- Thread switches cause a different register set to be used
- Hides memory latency caused by cache misses

True parallel processing uses multiple execution units
- The units run at the same time
- Each executing a separate thread
- These are called multiprocessor systems

Forms of Parallelism

Multi-core systems have 2 or more processors on a single chip
- required resources are shared:
  - Memory interface
  - Cache control
  - Interconnect systems

More cost effective than single sophisticated processors
- Increase performance without greater complexity
- Use lower clock rates, thus less power

Forms of Parallelism

In parallel systems, groups of processors work together

- *topology* defines the way the processors are interconnected

The collaborating processors must communicate

- There are two options:
  - Shared bus
  - Shared interconnection network

- Bus-based multiprocessors share memory
  - Processors can access another's memory directly

Forms of Parallelism

Message-passing multiprocessors use communication links such as ethernet or other proprietary high speed links

The degree of coupling differs for bus-based vs. message-passing
- Coupling is far higher for bus-based systems
- Bus-based systems have high bandwidth but are expensive
- Message-passing systems are less complicated

Bus-based systems are harder to scale
- access to the shared memory becomes harder to manage
- These are referred to a “tightly coupled”
- Versus the “loosely coupled” message-passing systems

Forms of Parallelism

Memory systems greatly affect multiprocessor performance

- Uniform memory access (UMA)
  - Equally accessible by all processors
  - Used with smaller tightly coupled bus-based systems

- NUMA (non-uniform memory access)
  - memory is not homogeneous
  - not all memory is equally accessible to all processors
  - Used in large message passing multiprocessors
  - Referred to a loosely coupled

Multiprocessor Performance

Using N processors may not provide a speedup of N
- Amdahl's law still applies
- Code must contain independent parts
- Each independent part can run on a different processor
- Sequential part does not benefit from extra processors

The size of the problem or task makes a difference
- Best to use more processors on larger problems
- Processors will be idle unless they have work to do

Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?

$$T_{\text{old}} = T_{\text{sequential}} + T_{\text{parallelizable}}$$

$$T_{\text{new}} = T_{\text{sequential}} + \frac{T_{\text{parallelizable}}}{100}$$

$$\text{Speedup} = \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{1}{\frac{T_{new}}{T_{\text{old}}}} = \frac{1}{\frac{T_{\text{sequential}}}{T_{\text{old}}} + \frac{T_{\text{parallelizable}}}{T_{\text{old}} * 100}}$$

Amdahl's Law

$$\text{Speedup} = \frac{1}{f_{\text{sequential}} + \frac{f_{\text{parallelizable}}}{100}}$$

$$\text{Speedup} = \frac{1}{(1 - f_{\text{parallelizable}}) + f_{\text{parallelizable}} / 100}$$

$$\text{Speedup} = \frac{1}{(1 - f_{\text{parallelizable}}) + f_{\text{parallelizable}} / 100} = 90$$

$$\text{Speedup} = \frac{1}{1 - 0.99 * f_{\,parallelizable}} = 90$$

$$f_{\,parallelizable} = \frac{1 - \frac{1}{90}}{0.99} = 0.999$$

So sequential part can only be 0.1% of the total.

Another example

An SMP system contains 8 processors.

A program consists of a startup sequential section
that produces results used in remaining parallel part

Desired speedup = 8/3
relative to executing the program on a single processor

Parallel part must be what percent of the total code?

Example

Let f = fraction of the code corresponding to parallel part
Based on definition of speedup:

$$\frac{1}{(1-f)+\frac{f}{8}} = \frac{8}{3} \longrightarrow 1 - \frac{7f}{8} = \frac{3}{8}$$

$$\frac{7f}{8} = \frac{5}{8} \longrightarrow f = \frac{5}{7} = 0.7143$$

71.43% of the code must be parallelizable

Strong versus Weak Scaling

Strong Scaling:

using more processors on a given size problem

Weak Scaling:

Increasing the number of processors with problem size

Good speedup is more difficult with strong scaling

Extra processors may sit idle unless problem size grows

Strong versus Weak Scaling

Example:

A program computes the sum of two 10element vectors and the sum of two 10x10 matrices

- Each addition takes 1 cycle
- 10 processors are available
- The vector sum is computed by 1 processor
- The matrix sum is split among 10 processors

The potential speedup is a factor of 10

Total time using 1 processor = 10 + 100 = 110 cycles

Time using 10 processors = 10 + 100/10 = 20 cycles

Speedup = 110/20 = 5.5

Achieves 55% of the potential speedup

Strong versus Weak Scaling

Example:

Suppose 40 processors are used instead

The potential speedup is a factor of 40

Total time using 1 processor = 10 + 100 = 110 cycles

Time using 40 processors = 10 + 100/40 = 12.5 cycles

Speedup = 110/12.5 = 8.8

Achieves only 22% of the potential speedup

Strong versus Weak Scaling

Example:
Suppose matrix size grows to 20x20
Total cycles using 1 processor = 10 + 400 = 410
cycles using 10 processors = 10 + 400/10 = 50
cycles using 40 processors = 10 + 400/40 = 20

Speedup with 10 processors = 410/50 = 8.2
Achieves 82% of the potential speedup

Speedup with 40 processors = 410/20 = 20.5
Achieves 51.25% of the potential speedup

The size of the problem affects the speedup

Strong versus Weak Scaling

Amdahl's Law for multi-processors

Let T1 be the execution time for a program on a single processor

f = fraction of time due to the parallel part split N ways

TN is the execution time using N processors

$$TN = \left[(1-f) + \frac{f}{N}\right] \mathrm{T1}$$

Adding extra cores only improves the parallel part

$$Speedup = \frac{T1}{TN} = \frac{1}{(1-f) + \frac{f}{N}} < \frac{1}{(1-f)}$$

Indicates an upper limit on the speedup (strong scaling)

Strong versus Weak Scaling

Gustafson's Law applies when N increases with problem size

- More processors can be used with larger workloads
- TN is the execution time using N processors
- f = fraction of TN used for parallel part on N-processor system

Gustafson's law states:

$$Speedup = (1 - f) + f * N$$

Speedup varies linearly with f (weak scaling)

As workload expands, parallel part becomes a larger fraction of TN

Flynn's Taxonomy

Classification based on instruction and data streams

- SISD – single instruction & single data stream
- MISD – multiple instruction streams & one data stream
- SIMD – single instruction stream & multiple data streams
- MIMD – multiple instruction streams & multiple data streams

- Flynn introduced this scheme in a 1972 paper

  Does not cover topology or interconnection techniques

SISD Systems

An instruction stream is a sequence of instructions

Fetched using a single program counter register

- SISD systems execute instructions sequentially
- Pipelining or multiple execute units may be used

Data operands are obtained one at a time from memory as a single stream of values

SIMD Systems

A single instruction stream is applied to multiple data

- Includes vector type instructions
- MMX multi-media extensions (Intel processors)
- A single control unit fetches and decodes instructions
- Multiple PEs act on separate operands in parallel
  - PEs are processing elements

Operands can be sub-sets of bits within a word or register

MISD Systems

Multiple instruction streams act on one data stream

- Few, if any, systems follow this model

A possible use is in highly fault tolerant systems

- Multiple processors produce results from the same data
- Majority vote determines the result used
- Example use is in fly-by-wire aircraft

MIMD Systems

Multiple instruction streams act on multiple data streams

- Multiple instructions are fetched and decoded in parallel
- Instructions may be vector or scalar type
- Each control unit directs one or more execute units (PEs)

Based on shared memory or on message passing

- May employ board level or chip level multiprocessors
- Chip level multiprocessors are also called multi-core
- Distributed systems require a high speed interconnect
- The interconnection scheme determines the topology

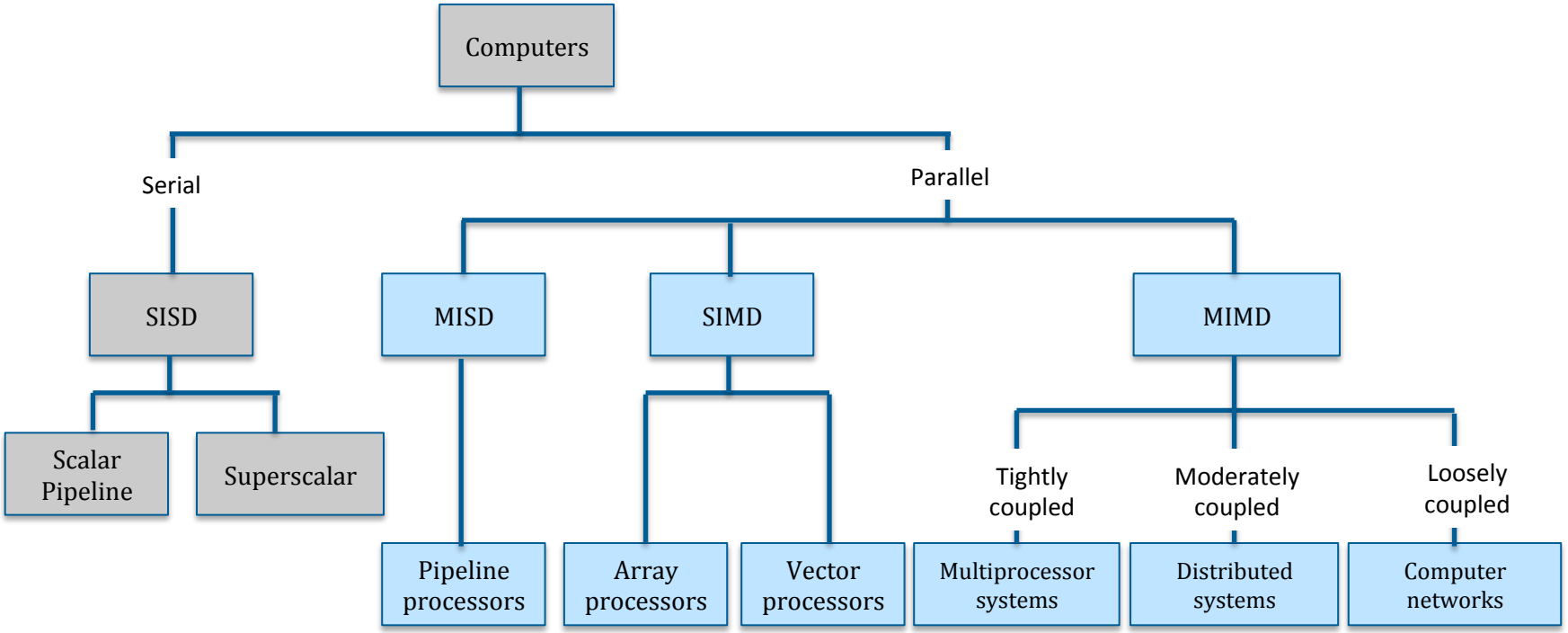
MIMD Systems

Collaborating programs can run on different processors

- One program can run on all processors of a MIMD system
- Conditional statements determine when different processors should execute different sections of the program
- This is known as SPMD (single program multiple data)

		Software	
		Sequential	**Concurrent**
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Core i7	Windows Vista Operating System running on an Intel Core i7

Flynn's Taxonomy

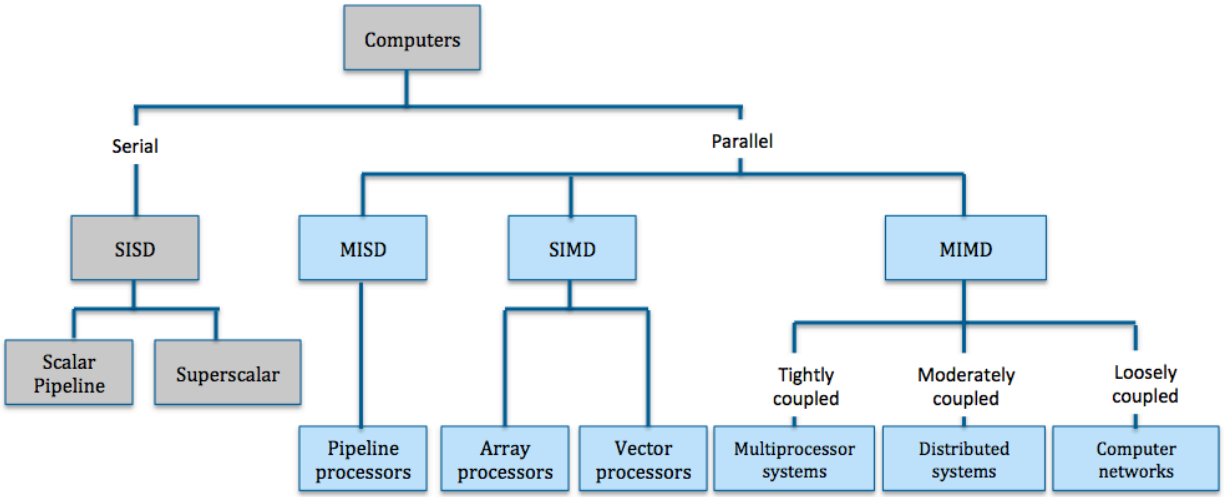


Multiprocessor Coupling

Multiprocessors can contain thousands of PEs

- Typically many fewer PEs are used due to coupling issues
- Goal is to efficiently increase performance
  - Avoid extremely high clock rates
  - Reduce heat and power dissipation

Coupling describes the degree of interconnection



Multiprocessor Coupling

- Loosely coupled systems employ message passing
  - Exchange data slowly relative to CPU clock speed
  - Connecting via the internet is one extreme example
  - Granularity of data is high (entire files)

- Tightly coupled systems use shared memory
  - Provide high data bandwidth and low latency
  - Use fine grained granularity
  - Processors can operate on the same data structure
    - For example: on different elements within an array

Multiprocessor Coupling

- There are issues related to sharing memory
  - Synchronization
  - Cache consistency
  - Access order

- These issues will be addressed later

Vector Processors

A vector computer is a SIMD machine

- Single vector instructions operate on multi-element data items
- Commands are broadcast to the processing elements (PE)
- PE's operate in lock-step fashion performing the same operation

Example: Z = s*X + Y where Z, X and Y are arrays or vectors
s is a scalar constant or variable

Intel-based PC's have MMX, SSE & AVX instructions

- Multi-media extensions (integer)
- Streaming SIMD extensions (floating point)
- Advanced Vector Extensions (AVX)
  - Operates on four 64-bit floating point numbers in parallel

Vector Processors

Only supercomputers had vector instructions in the past
the Cray-1 designed by Seymour Cray in the 70s

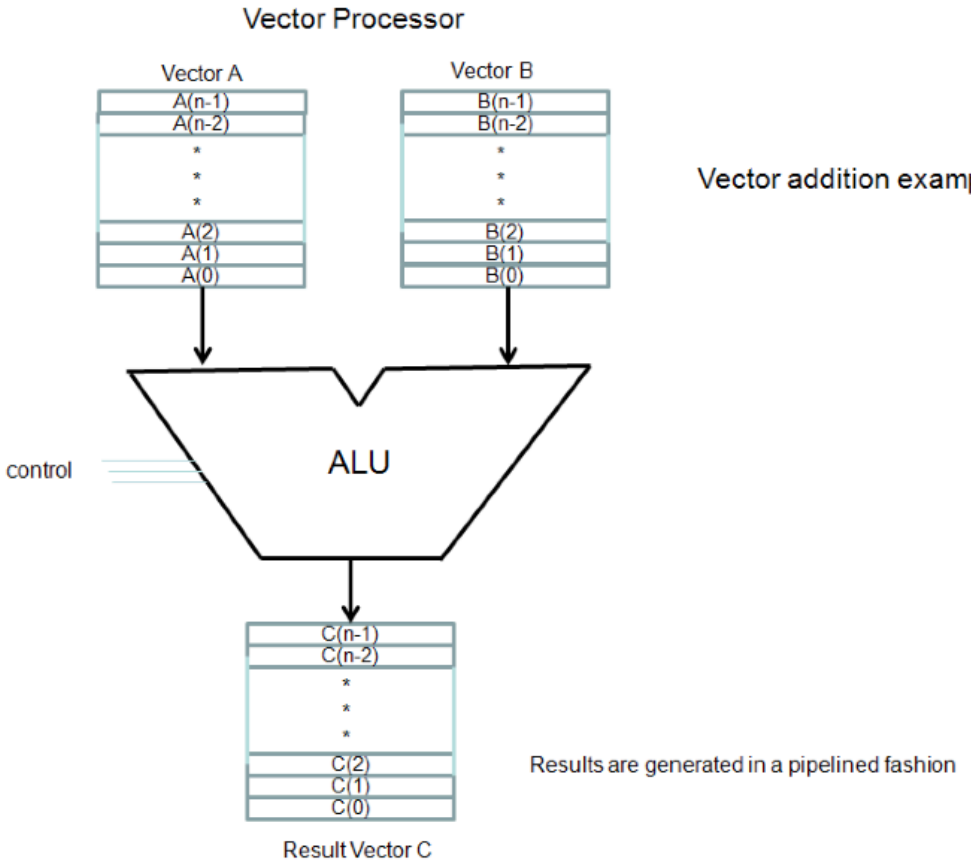
- Highly pipelined functional units
- Data get streamed to and from vector registers
  - Data collected from memory into registers
  - Results stored from registers to memory

Basic idea:

- Read sets of data elements into “vector registers”
- Operate on those registers
- Disperse the results back into memory

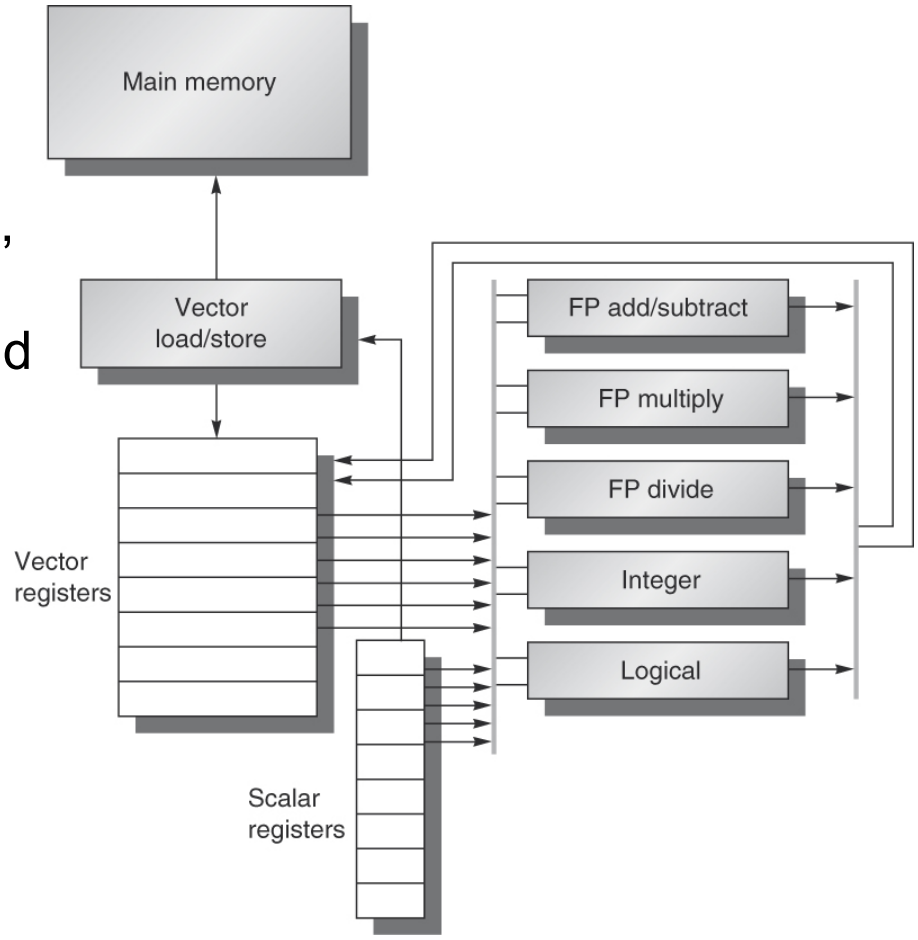
Vector Processors

High memory bandwidth is required to quickly fill registers



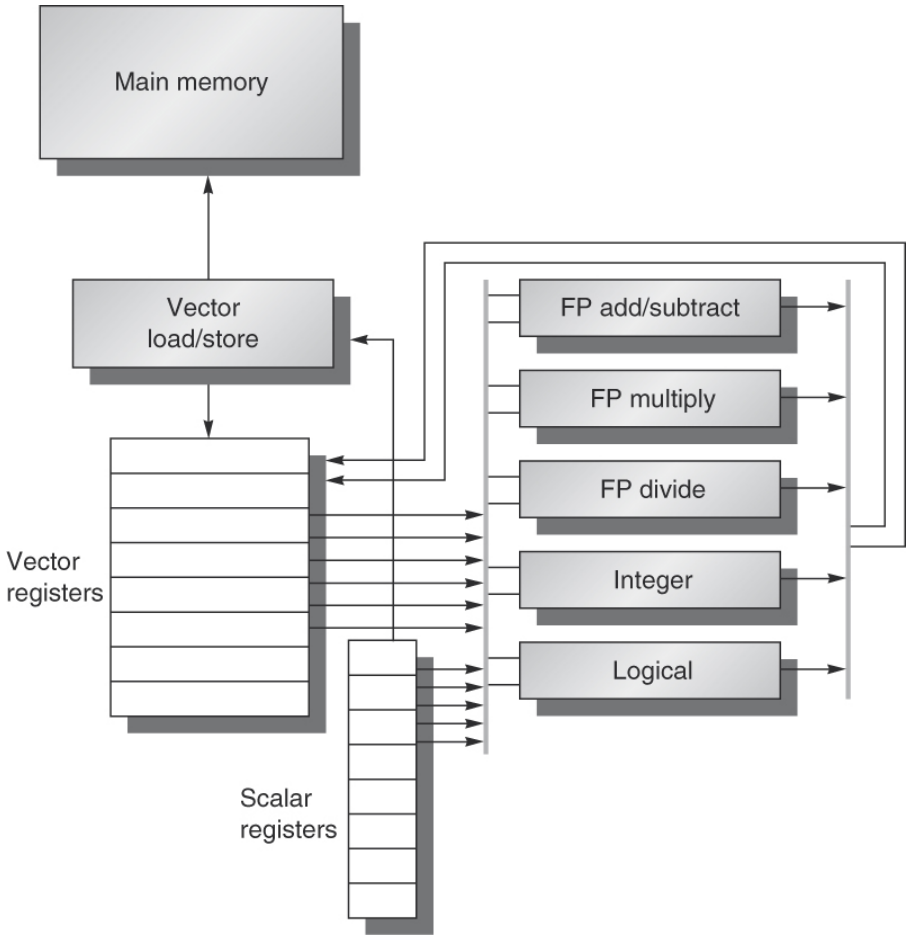
Example VMIPS

- Vector registers
  - Each register holds a 64-element, 64 bits/element vector
  - Register file has 16 read ports and 8 write ports



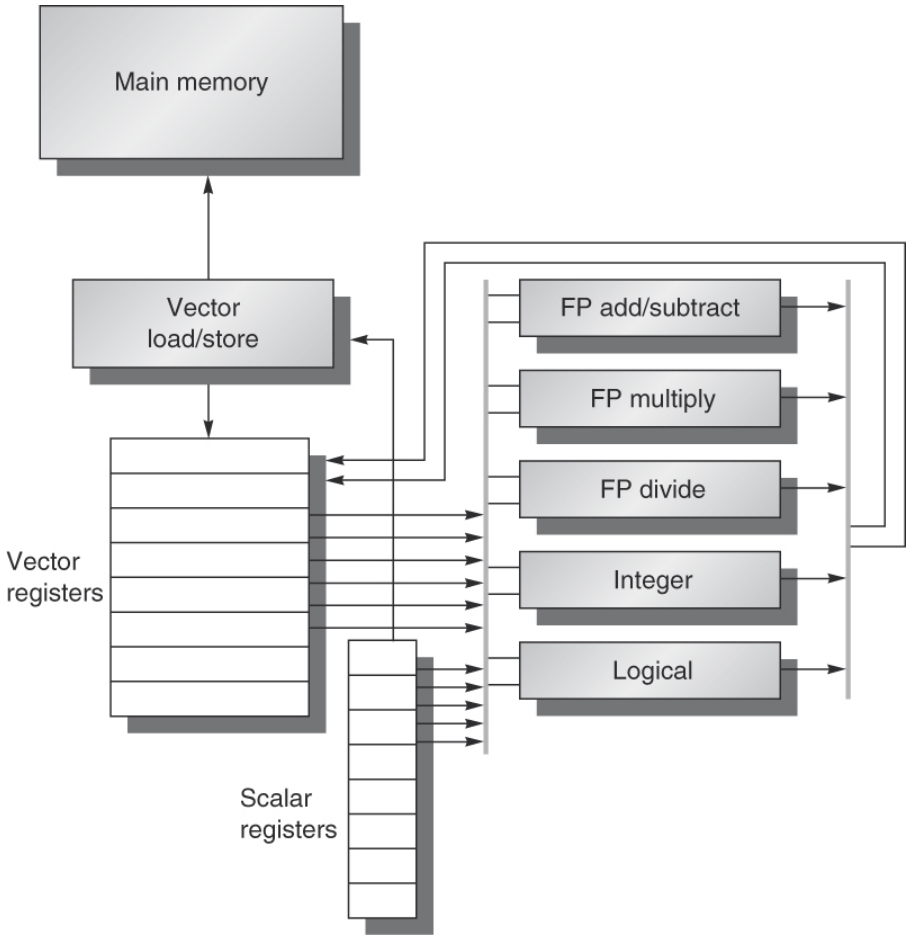
Example VMIPS

- Vector functional units
  - Fully pipelined
  - Data and control hazards are detected



Example VMIPS

- Vector load-store unit
  - Fully pipelined
  - Words move between registers
  - One word per clock cycle after initial latency
- Scalar registers
  - 32 general-purpose registers
  - 32 floating-point registers



Example VMIPS Instructions

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

```
# Scalar Code
  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
```

```
# Vector Code
  LI      VLR, 64
  LV      V1, R1
  LV      V2, R2
  ADDV.D  V3, V1, V2
  SV      V3, R3
```

Example VMIPS Instructions

- Example: DAXPY

```
L.D       F0,a      ;load scalar a
LV        V1,Rx     ;load vector X
MULVS.D   V2,V1,F0  ;vector-scalar mult
LV        V3,Ry     ;load vector Y
ADDVV     V4,V2,V3  ;add
SV        Ry,V4     ;store result
```

- In MIPS Code
  - ADD waits for MUL, SD waits for ADD
- In VMIPS
  - Stall once for the first vector element, subsequent elements will flow smoothly down the pipeline.
  - Pipeline stall required once per vector instruction!

Example VMIPS Instructions

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- VMIPS functional units consume one element per clock cycle
  - Execution time is approximately the vector length

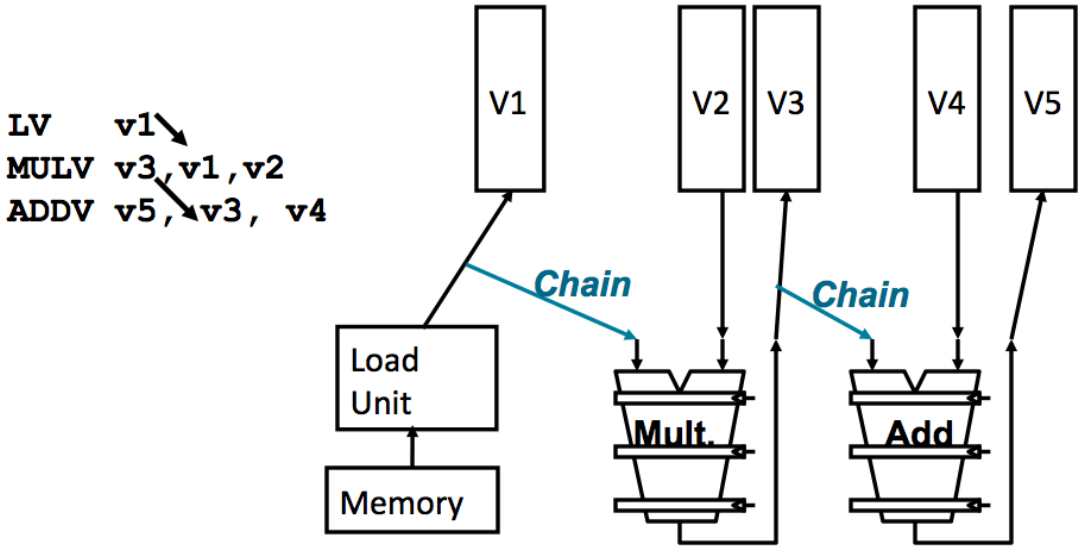
Vector Chaining

Input operands are used as soon as they arrive from memory

- Avoids stalling until the entire vector register is filled

Results can be sent from one functional unit directly to another

- A vector version of register bypassing and forwarding



Vector Chaining

Chaining overlaps the loading of operands into vector registers
- The next operands are read in parallel with the use of the previous set

The vector length register (VLR) specifies number of reads

Vector elements do not have to occupy adjacent memory cells
- Unlike with multi-media extensions (MMX, SSE and AVX)
- *Stride* = amount by which elements are separated in memory

Results can be chained back to memory as they are produced

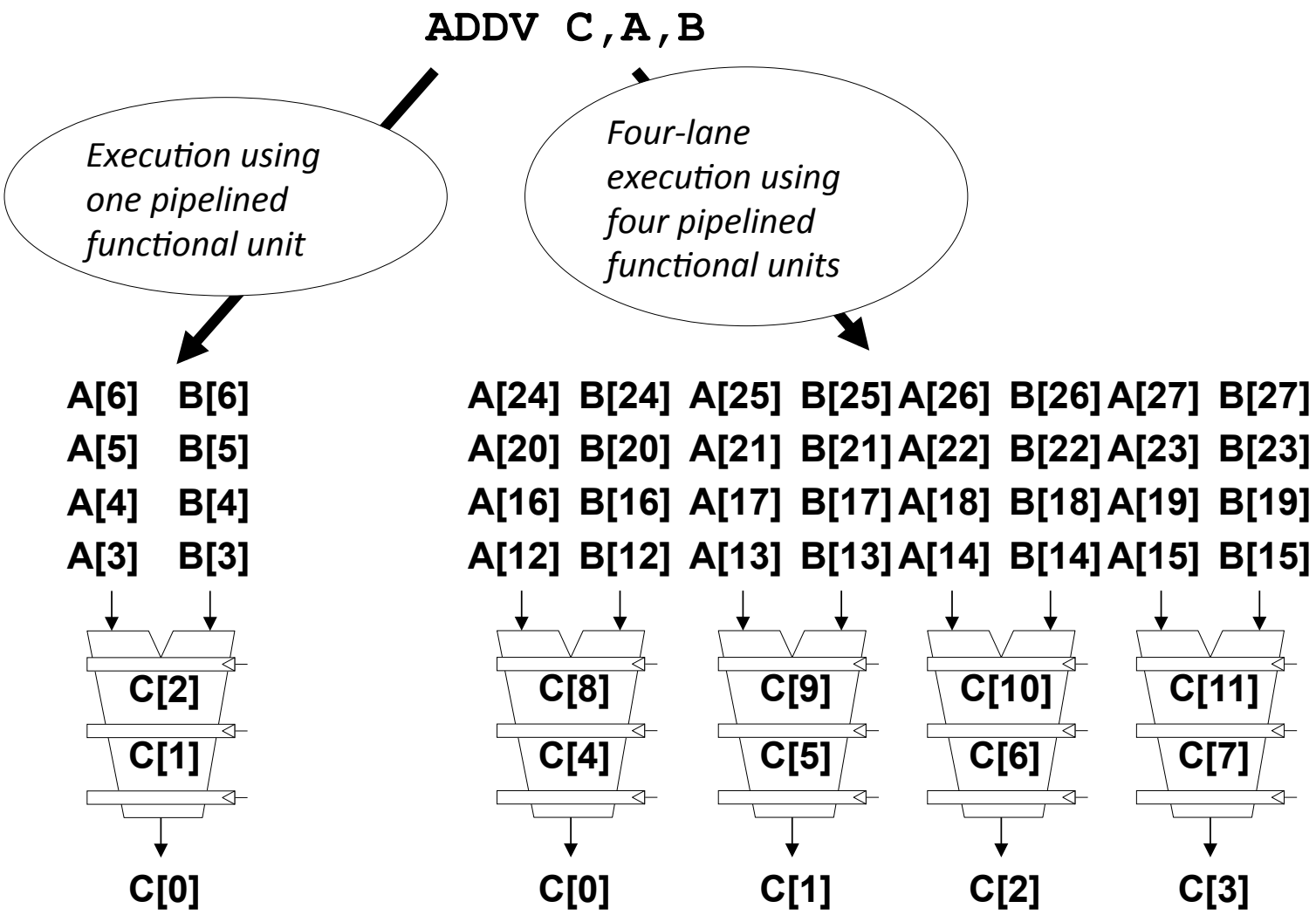
Scatter/ gather

Some vector processors use registers to locate data items
- vector registers need not be loaded from adjacent locations
- These indexed loads are called a *gather* operation

Indexing via registers can also be used to store vector results
- Elements in vector registers are dispersed to non-adjacent locations
- This is known as a *scatter* operation

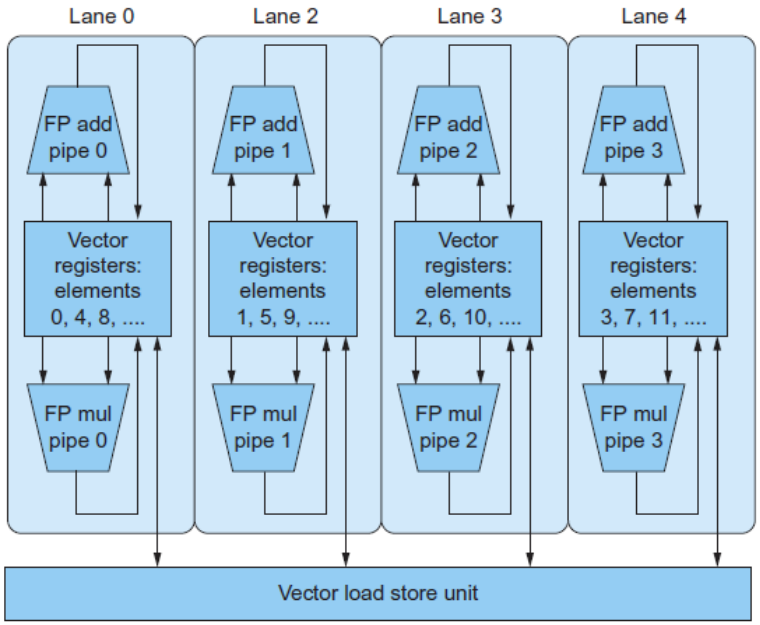
Strip mining compensates for mismatch in vector register length
- E.g. if vector is of length 60 and register is of length 32
- Vector register is loaded with first 32 elements
- And VLR (vector length register) is adjusted to load the next 28

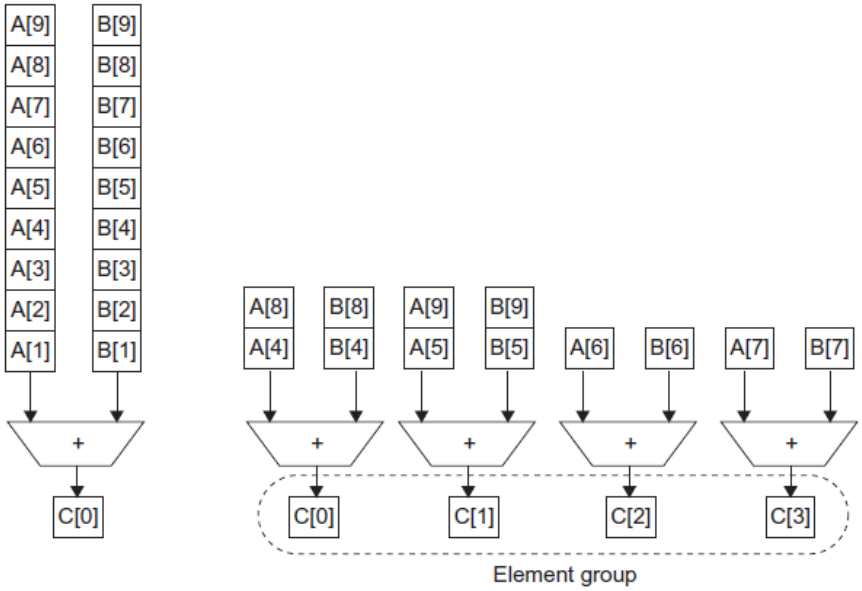
Example Vector Addition



Vector Processors

- Element n of vector register A is "hardwired" to element n of vector register B
  - Allows for multiple hardware lanes
  - No communication between lanes
  - Little increase in control overhead
  - No need to change machine code

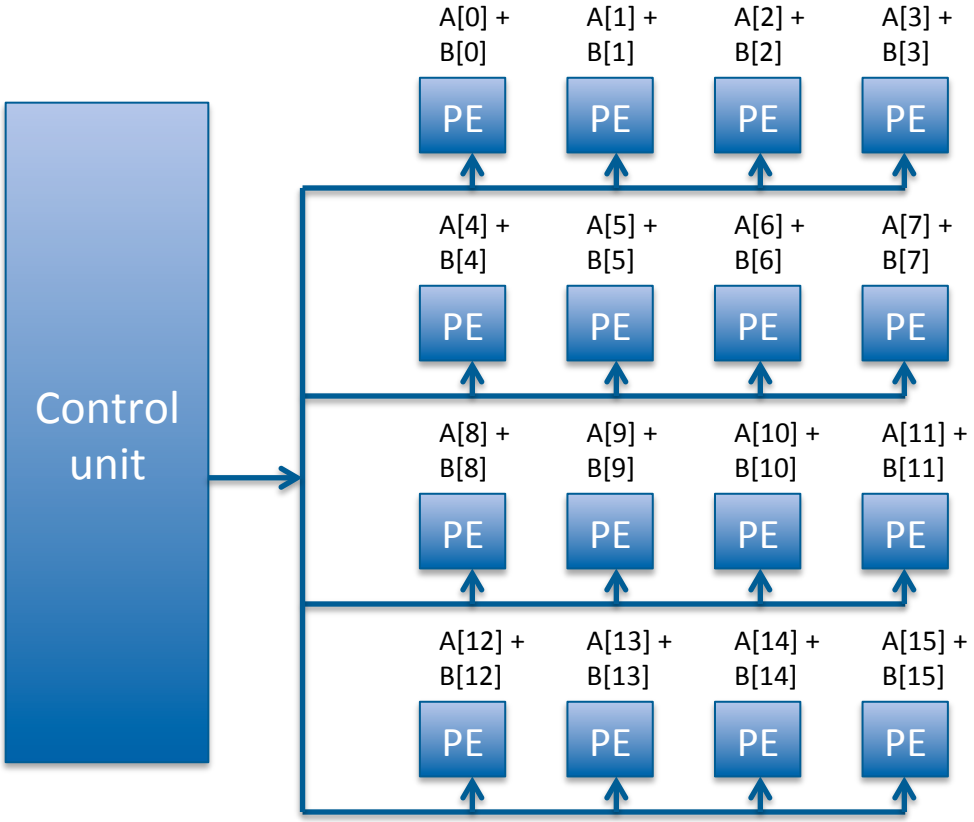




Adding more lanes allows designers to tradeoff clock rate and energy without sacrificing performance!

Array Processors

- An alternative in an array processor (SIMD)
- Separate processing elements could add corresponding array elements



One control unit broadcasts the same command to multiple processing elements that operate in lock-step fashion.

Symmetric Multiprocessors

Symmetric Multiprocessors are said to be tightly coupled

- Also called shared memory multiprocessor
- A type of MIMD system

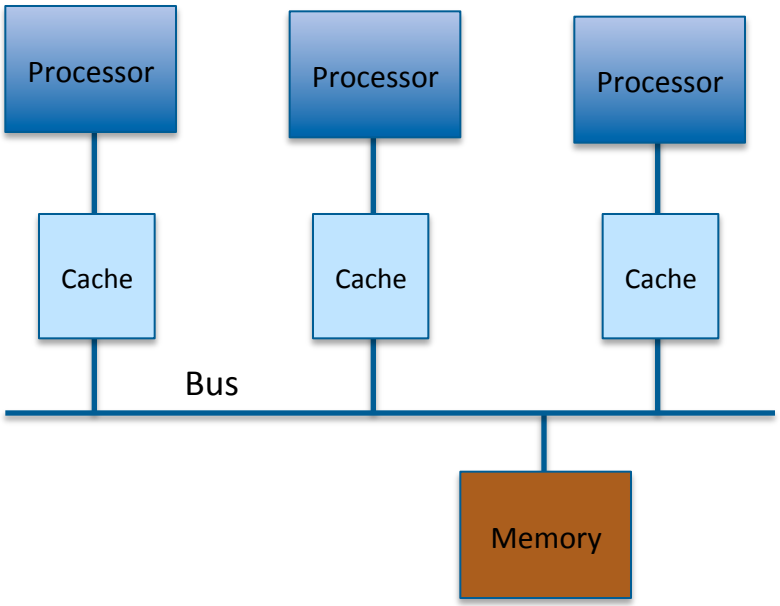
An SMP architecture treats all processors equally

- *Symmetric* implies the processors are logically interchangeable
- The OS divides and distributes the work to processors

Programs can be written to run faster on SMPs

Programs optimized for SMP will suffer if run on uniprocessor

Symmetric Multiprocessors

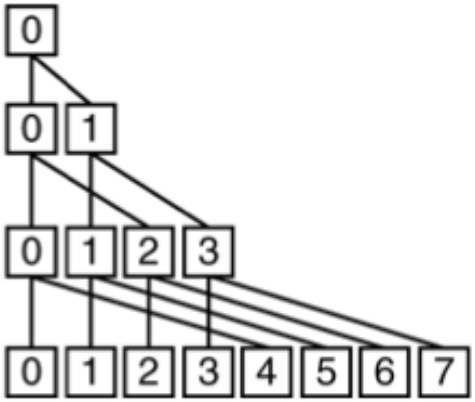


SMPs use a common shared bus

- The bus may become a bottleneck
- Processors must make good use of their internal caches
- Cache misses cause stalls and contention for the bus

Example: Sum Reduction on SMP

- To illustrate the idea, assume an 8-element array
- Assume there are 4 processors
- Each will add 2 elements




```
Sum[P0] = A[0] + A[1]
Sum[P1] = A[2] + A[3]
Sum[P2] = A[4] + A[5]
Sum[P3] = A[6] + A[7]

Then use half the processors (2)
Sum[P0] = Sum[P0] + Sum[P2]
Sum[P1] = Sum[P1] + Sum[P3]

Finally use one processor: Sum[P0] = Sum[P0] + Sum[P1]
```

Example: Sum Reduction on SMP

- Sum 100,000 numbers on 100 processor UMA
  - Each processor has ID: 0 ≤ Pn ≤ 99
  - Partition 1000 numbers per processor
  - Initial summation on each processor

```
sum[Pn] = 0;
  for (i = 1000*Pn;
      i < 1000*(Pn+1); i = i + 1)
    sum[Pn] = sum[Pn] + A[i];
```

```
Sum[0] = A[0] + … + A[999]
.
.
.
Sum[99] = A[99000]+ … + A[99999]
```

Example: Sum Reduction on SMP

- Now need to add these 50 partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, …
  - 50, then 25, then 12, then 6, then 2, then 1
  - Using ½ may yield an odd number
    - P0 takes care of the left over value
  - Need to synchronize between reduction steps

Example: Sum Reduction on SMP

```
half = 100;
do
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
while (half > 1);  // exit with final sum in Sum[0]
```

synch() insures that all required partial sums have been produced

Private variables, such as *half* or a loop index, are local to each processor

Clusters

- Clusters are loosely coupled
- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- High availability, scalable, affordable

SMPs versus Clusters

- SMPs
  - Processors are identical
  - Controlled by a single operating system
  - Cost to administer is about the same as a uniprocessor
  - Communicate via memory bus
  - More difficult to scale
  - Higher cost to purchase

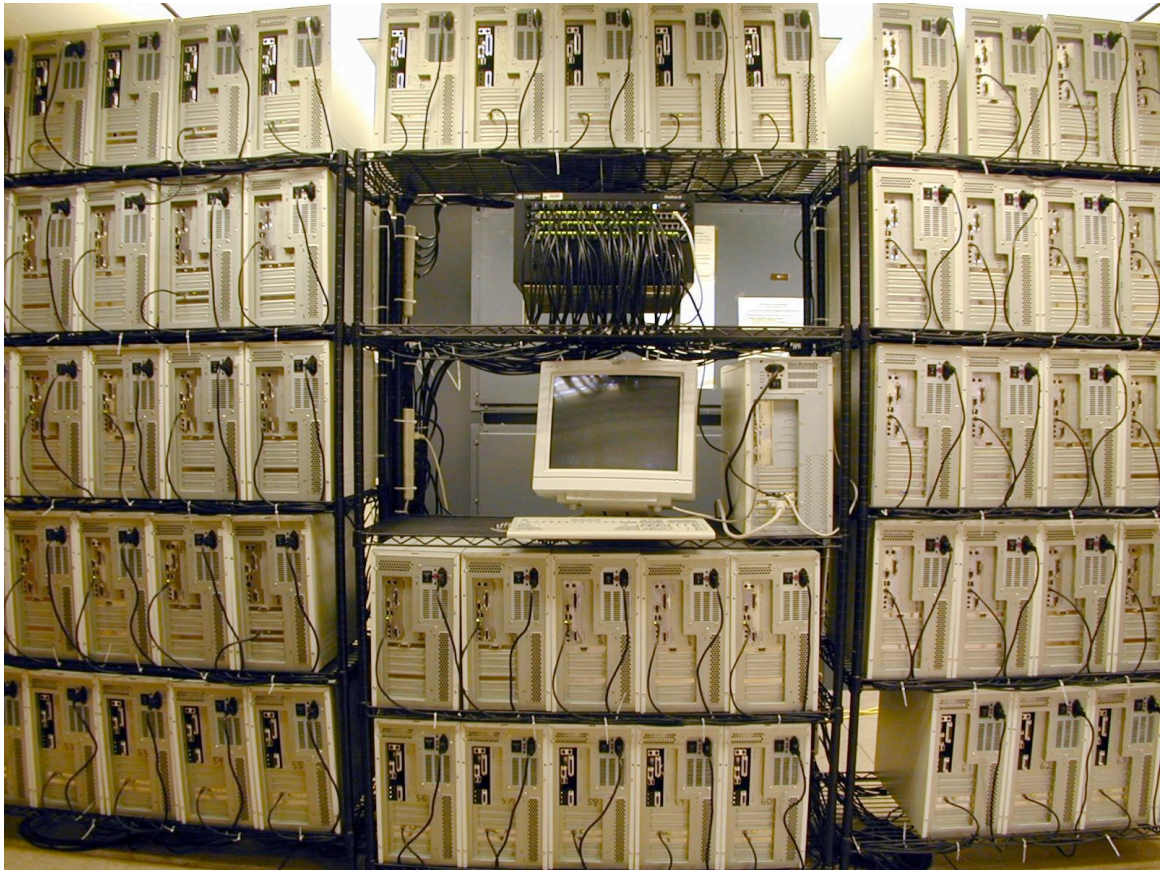
SMPs versus Clusters

- Clusters
  - Each node is a complete computer
  - Nodes can employ different processors
  - Cost to administer N-node system is about the same as for N separate computer systems
  - Communicate via I/O bus at a slower rate
  - Easier to scale
  - Nodes may be low cost COTS (commodity off the shelf) computers

Beowulf Cluster

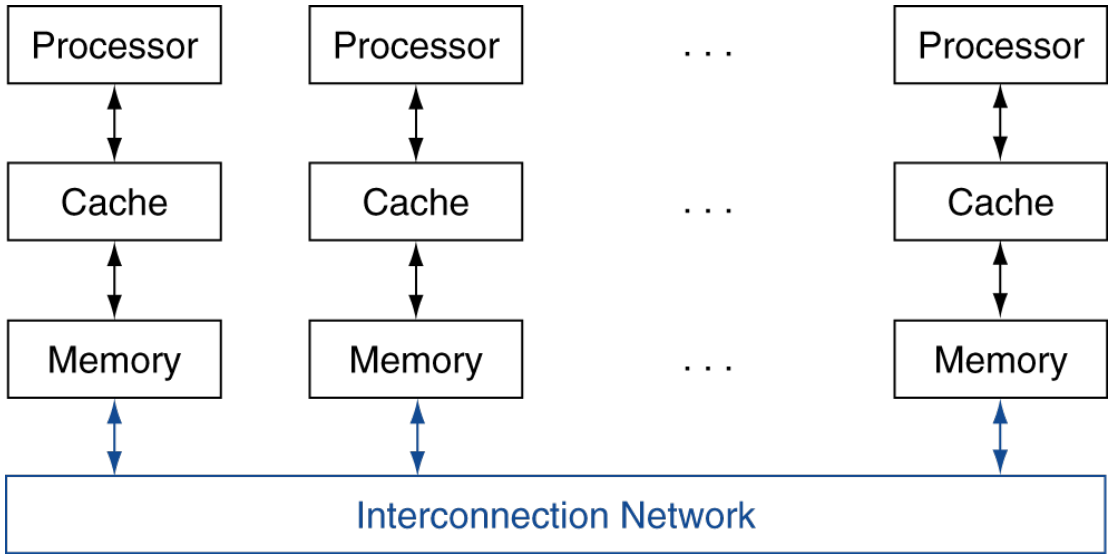
- Identical commodity-grade computers networked into a LAN
- Originally referred to a computer built in 1994 by Thomas Sterling and Donald Becker at NASA
- Normally use a unix-like operating system
- One server node and one or more client nodes connected via ethernet

Beowulf Cluster



Message Passing

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



Sum Reduction via cluster

- Sum 100,000 on 100 processors
- First distribute 1000 numbers to each
  - The do partial sums

```
sum = 0;
for (i = 0; i<1000; i = i + 1)
   sum = sum + AN[i];
```

Sum Reduction via cluster

- Reduction
  - Half the processors send, other half receive and add
  - Then ¼ send, & ¼ receive and add, …
  - Send/receive also provide synchronization
  - Assumes send/receive take similar time to addition

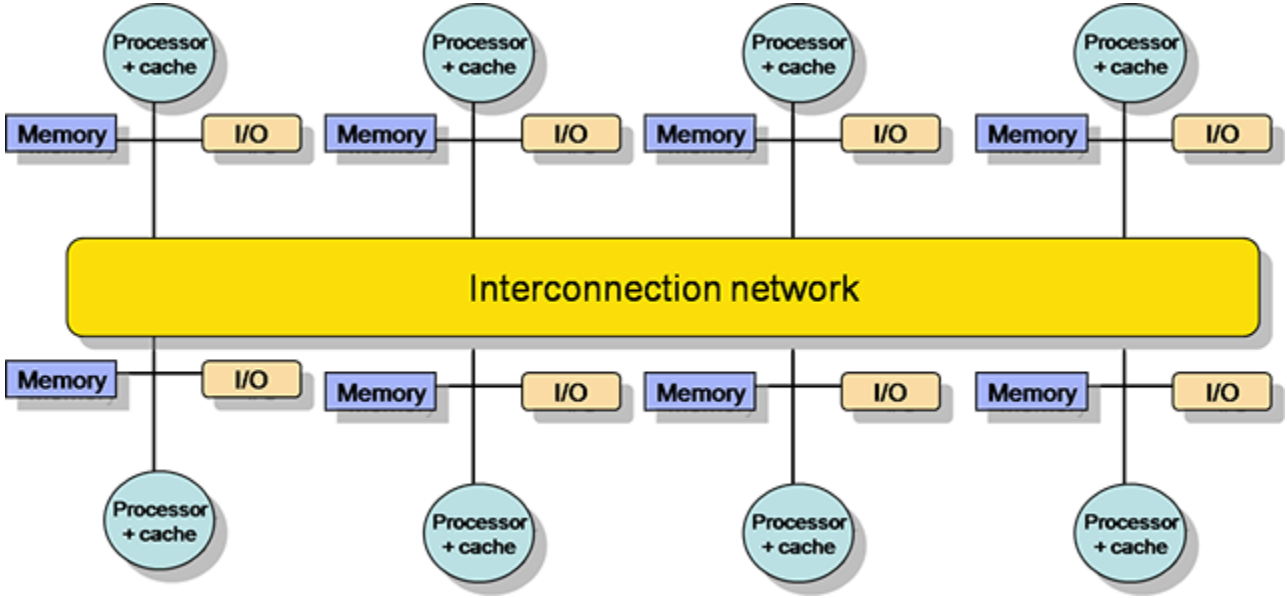
Example: Sum Reduction

- Given send() and receive() operations

```
limit = 100; half = 100;/* 100 processors */
repeat
  half = (half+1)/2; /* send vs. receive
                        dividing line */
  if (Pn >= half && Pn < limit)
    send(Pn - half, sum);
  if (Pn < (limit/2))
    sum = sum + receive();
  limit = half; /* upper limit of senders */
until (half == 1); /* exit with final sum */
```

NORMA Systems

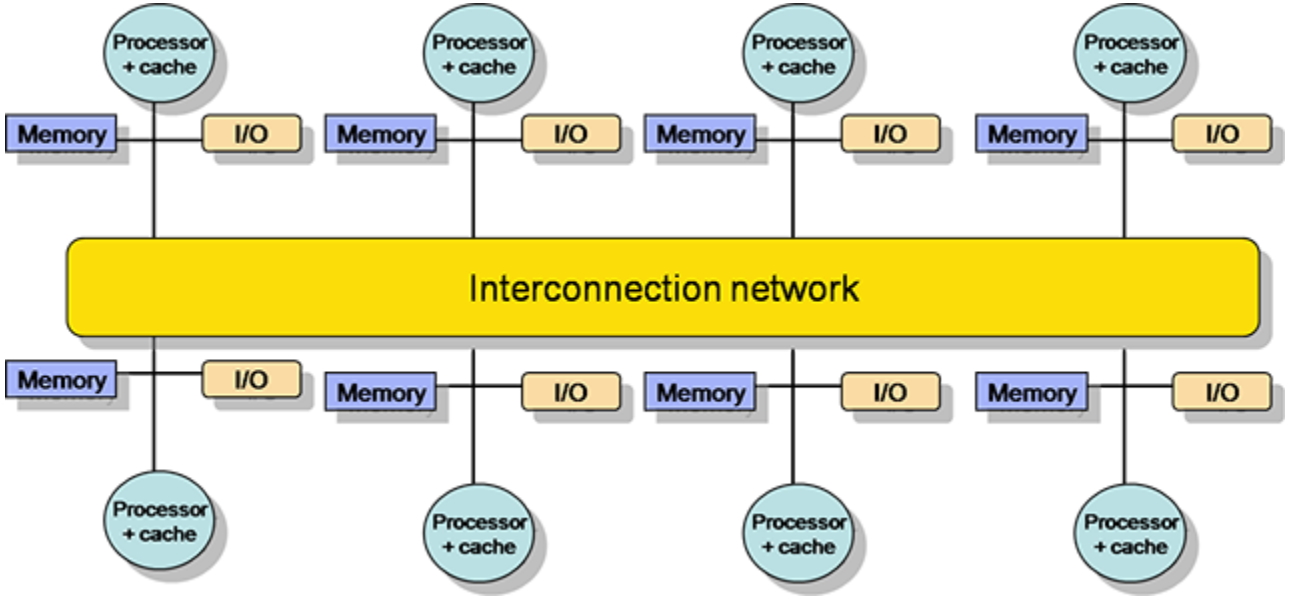
NORMA (no remote memory access) systems



Each processor has a separate private memory & address space

NORMA Systems

Processors only have direct access to their local memory



Message exchange is used to obtain data from remote modules

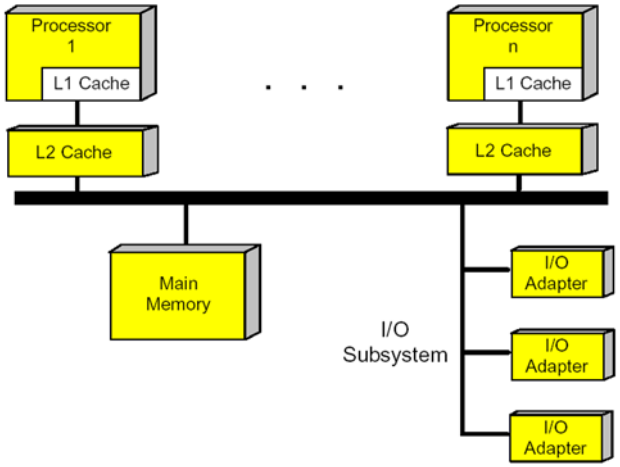
Clusters fit this loosely coupled model

These are easier to scale than the SMP tightly coupled counterpart

UMA on SMPs

Using a single global memory unit

- Uniform memory access (UMA) for all processors



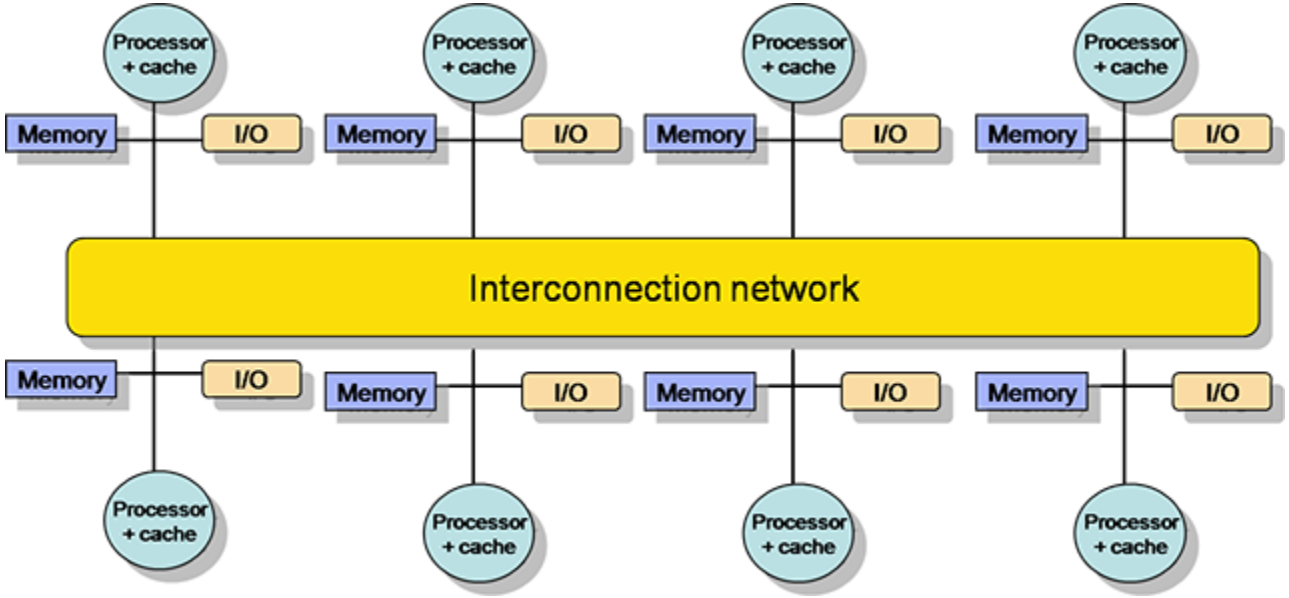
Only one processor at a time can access memory

- Competition for the CPU-to-memory bus will be high
- Cache can be used to reduce the conflicts
- This would be too limiting

NUMA Systems

Distributed memory modules work better

- Non-uniform memory access (NUMA) system

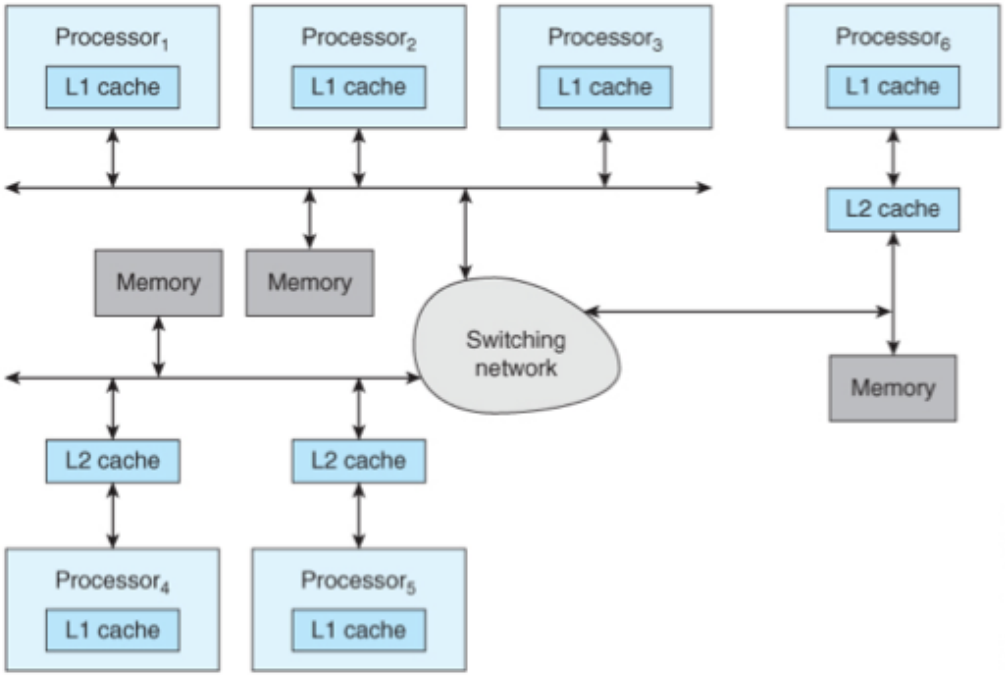


Local module access is faster than to a remote module

- CPU addresses map to a particular memory module
- Single shared address space spans all modules
- Each processor has a cache system

NUMA Systems

Another design for a NUMA system is shown below:



Memory is accessed less often when cache is used

But shared data must be kept consistent (i.e., coherent)

Coherency Requirements

Correct program order must be preserved

- Each processor must read the most recent version of data

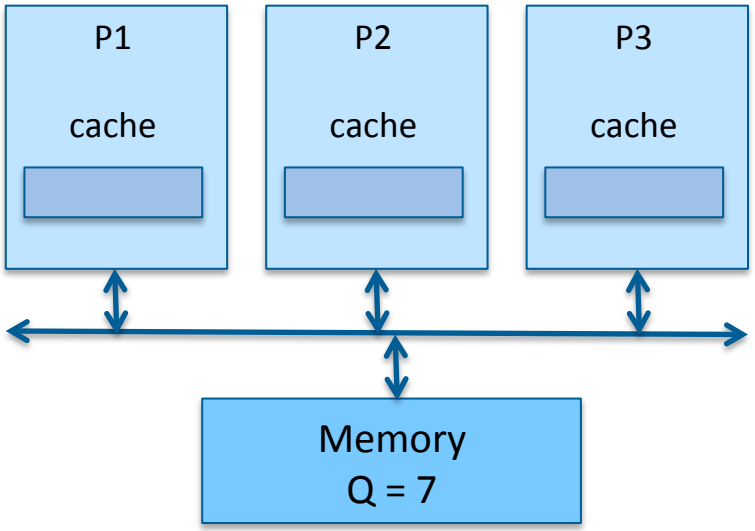
All writes must be visible to all processors

- If P1 writes x and then P2 reads x, P2 must obtain value written by P1

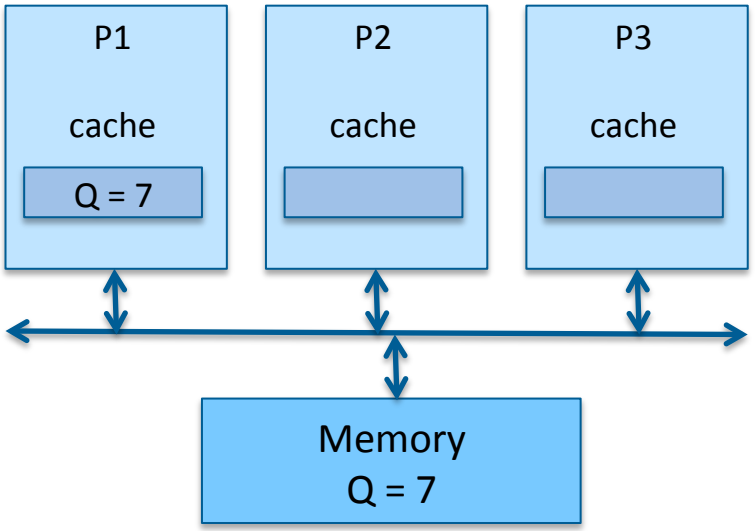
Order must be preserved

- Example: P1 sets x to 1, P2 reads x and if x==1, sets it to 2
- If P3 then reads x, it should see 2 and not the old value 1

Coherency Example

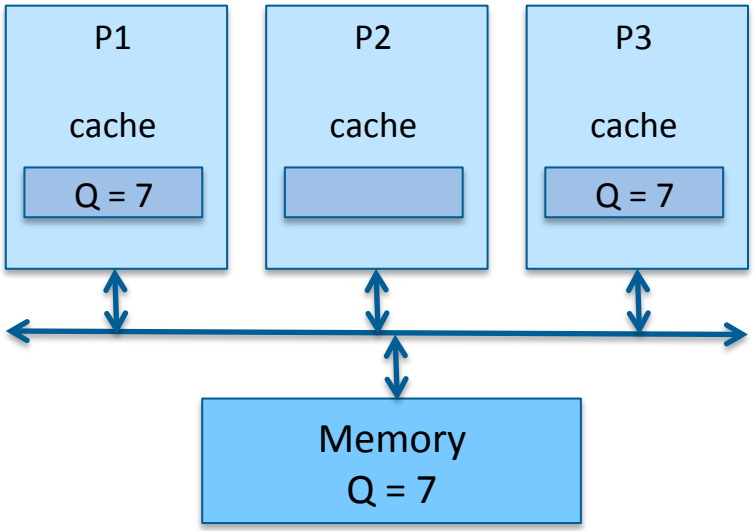


a) Initially Q = 7 in memory

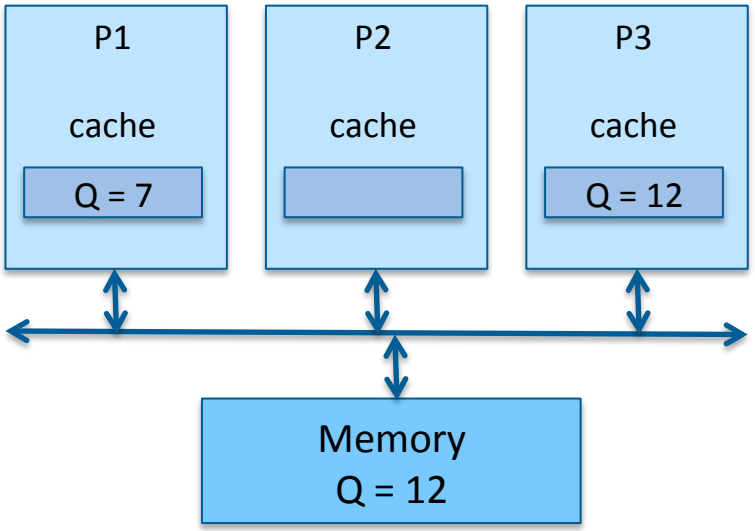


b) P1 reads Q and caches it locally

Coherency Example

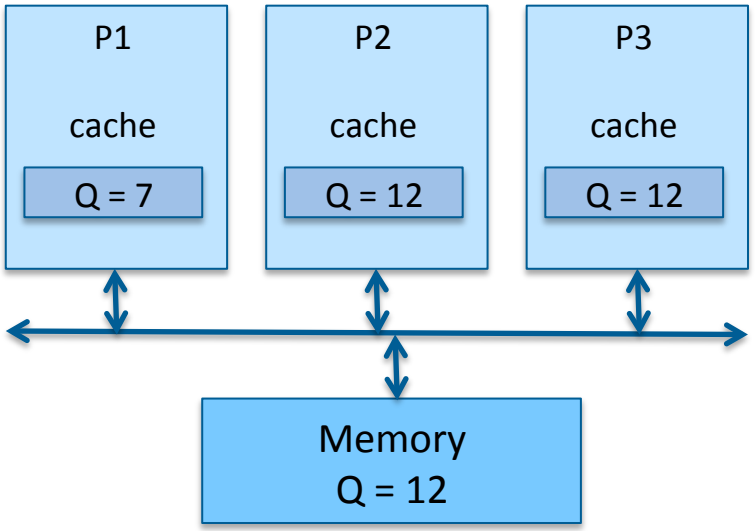


c) P3 reads Q from memory and caches it.

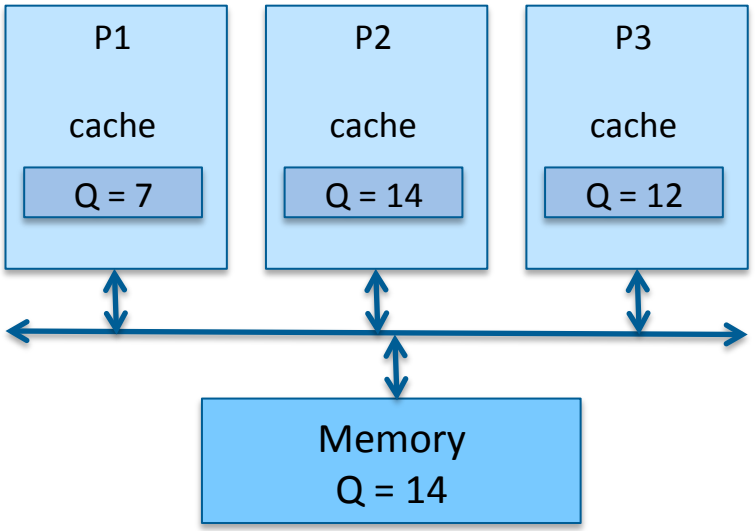


d) P3 writes Q=12 to cache and to memory

Coherency Example



e) P2 reads Q from memory and caches it



f) P2 computes Q + 2, caches it locally, and writes it back to memory

There are now three different cached values of Q
This happened because cache coherency was not maintained.

Cache Coherency

MESI is a protocol use by Intel and others

- Each cache line can be in one of 4 states:
- M – modified (in one cache & was altered since read from memory)
- E – exclusive (in one cache & matches what's in memory)
- S – shared (in at least 2 caches and matches what's in memory)
- I – invalid (never filled or is outdated)

Lines change state based on accesses that are made

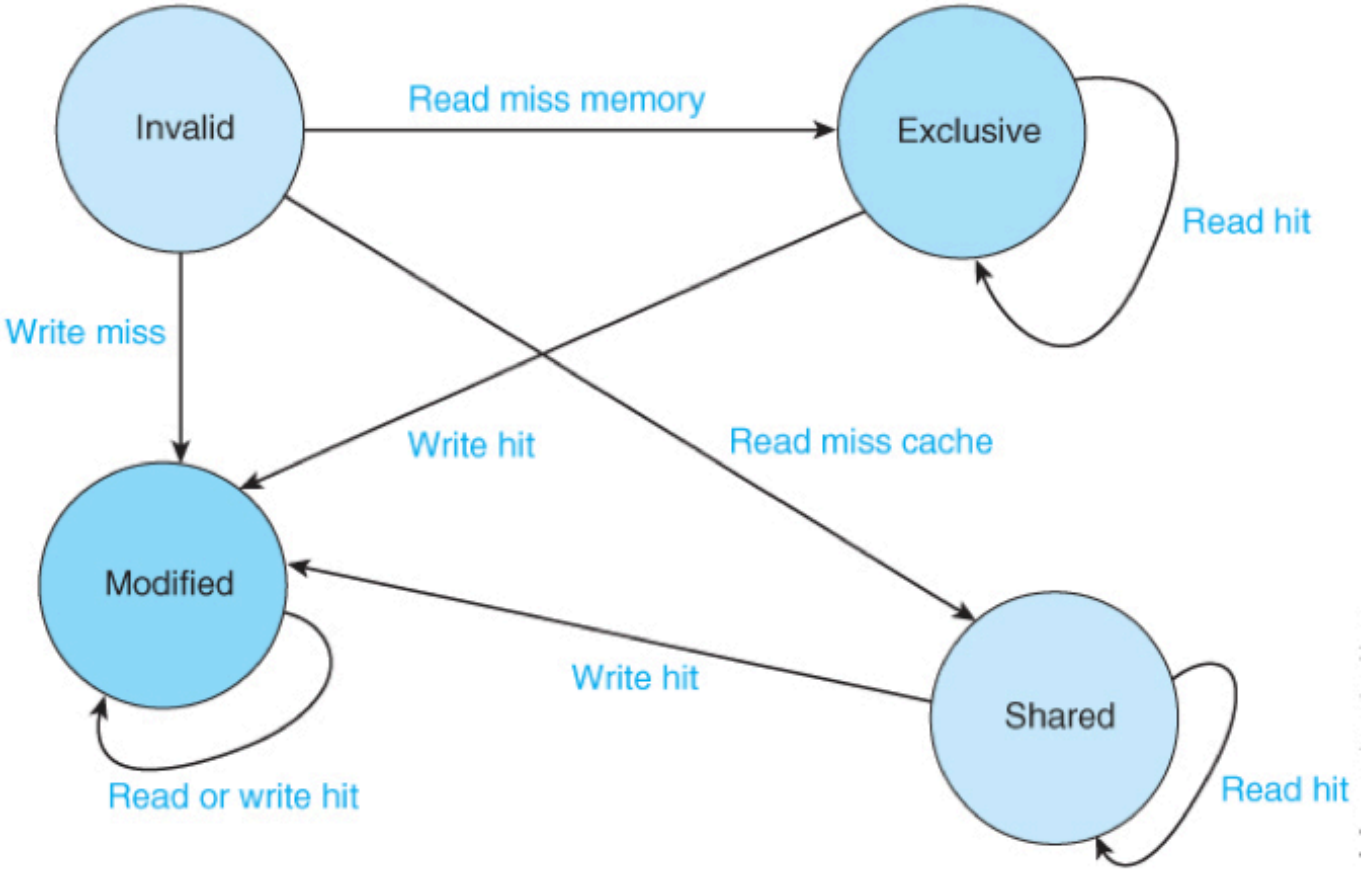
- From local processor or from other processors over the bus
- Cache controller listens (snoops) on the bus for accesses
- A *snoopy* protocol

Snoopy Protocol

- Writes are broadcast on the bus
  - All caches observe the write
  - Writes may invalidate copies of lines in other caches
- Does not scale beyond about 128 processors
  - The bus traffic due to snooping becomes too high
  - Exclusive and modified states reduces traffic
    - Writes to modified or exclusive lines need not be broadcast

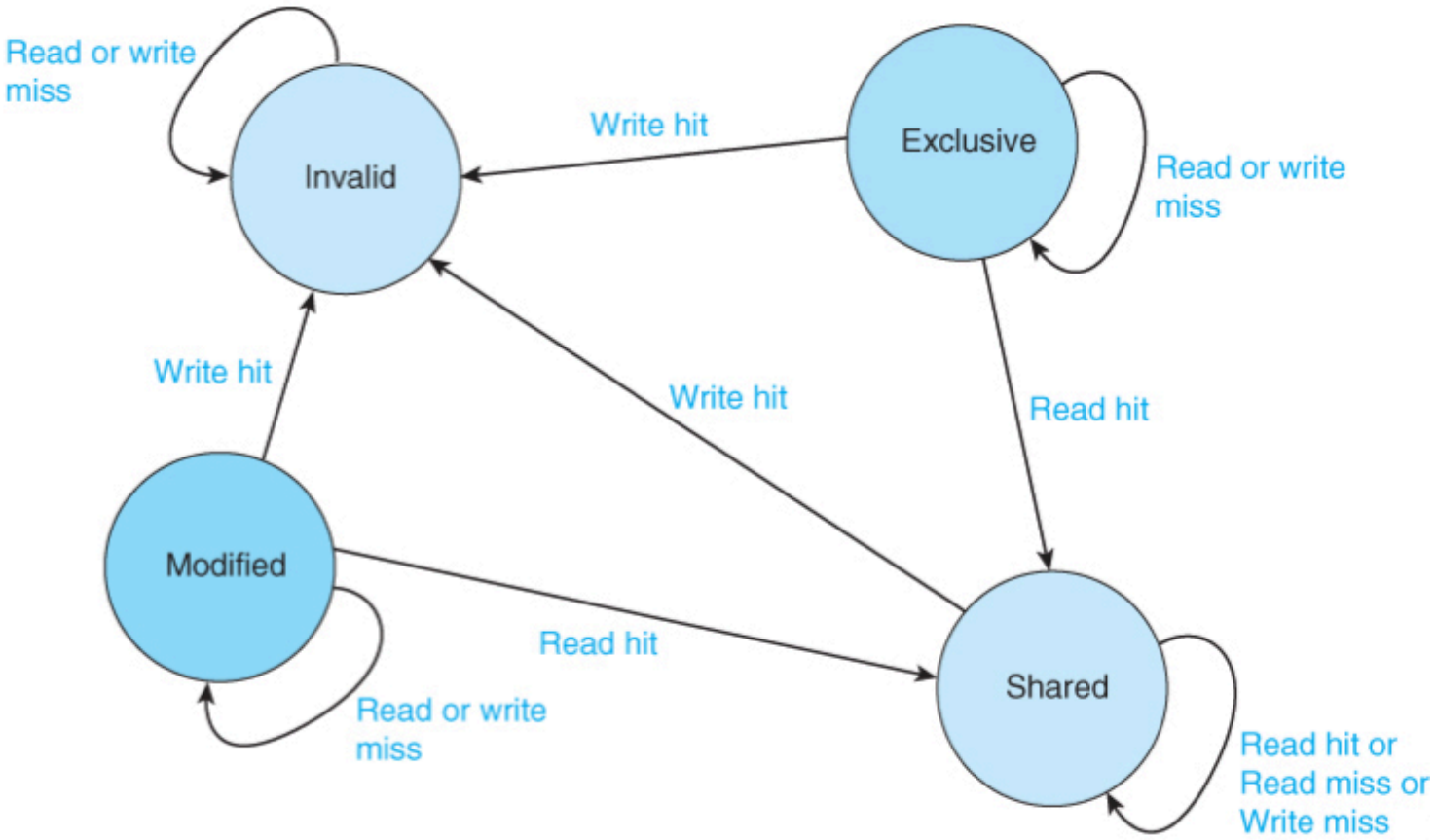
Snoopy Protocol

MESI state diagram as seen from the local CPU bus



Snoopy Protocol

MESI state diagram as seen from the system bus



MESI State Change

Bus Transaction	Action by local cache
Read hit	E➔ S, M➔ S or no change if S
Read miss	no change
Write hit	E ➔ I, S ➔ I, M ➔ I
Write miss	no change

Directory Based Protocol

Snoopy caches work well when connected to a single bus

Large shared-memory multiprocessors use interconnects

- These are networks such as rings or meshes
- Broadcasting cache operations to all processors would be inefficient

Cache directories can be used as an alternative to snooping

- Each memory module would have a directory
- This approach scales better than snooping on a shared bus

Directory Based Protocol

Directories identify which nodes contain a copy of a block

- The state of each cache line containing a copy is recorded

- Accesses to a module are intercepted by the directory

- The directory determines the action to take:
  - Reads are forwarded to the cache containing the copy
  - Writes are sent just to the nodes whose copies are affected
  - Misses cause the memory module to be accessed

Multithreading

Superscalar and VLIW systems are based on ILP

- Instruction level parallelism
- The processor fetches and executes bundles of instructions
- Hardware executes as many instructions as possible in parallel
- Compilers combine instructions into long words on VLIW systems

Thread level parallelism overlaps streams of instructions

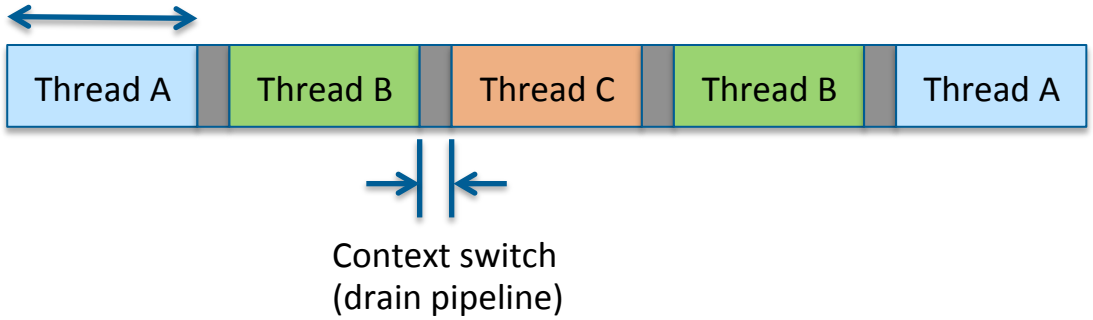
- Processes or tasks are split into separate threads
- When one thread stalls, control is switch to another
- This hides latencies due to cache misses or I/O
- All threads appear to be running at the same time

Tasks versus Threads

- Tasks & threads differ in scale or granularity
  - Tasks involve longer streams of instructions than do threads
    - Multitasking involves context switches and more overhead
    - Must complete pending instructions, save registers & flush cache
    - Triggered, for example, by a page fault
- Thread switching is more efficient
  - Hardware has multiple register sets
    - no saving or restoring is needed when switching
- Multithreading can be coarse-grained or fine-grained

Coarse-grained

- Switch occurs after a group of instructions are executed
  - Pipeline is drained each time a switch takes place



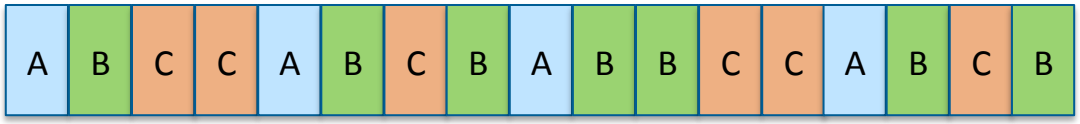
Expiration of a time slice or having to wait for I/O to complete trigger the switching

Fine-grained

Switch occurs at the end of each cycle

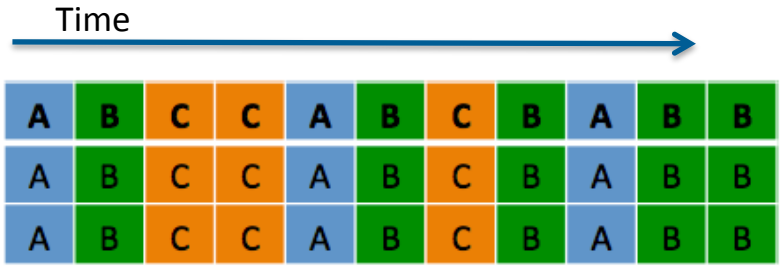
Each thread has its own set of registers

No need to drain the pipeline



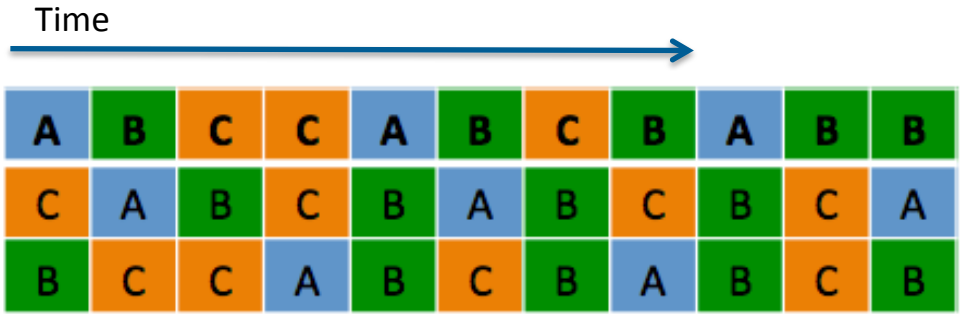
- Instructions from multiple threads are interleaved within the pipeline
- Having 2 or more processors allows several threads to execute in parallel
- To be efficient, there must be enough threads to keep the processors busy

Multiprocessors may also be superscalar



(each column corresponds to a clock cycle)

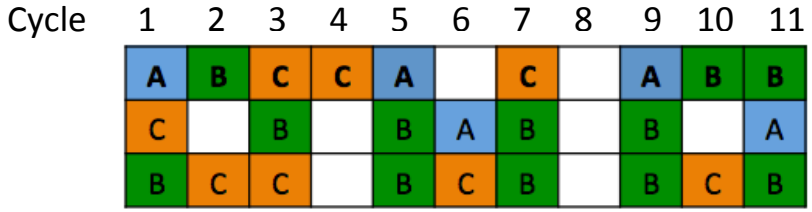
Fine-grained multithreading on a processor with three execute units which are available to only one thread at a time



Fine-grained simultaneous multithreading (SMT)
Execute units are available to all threads

Some execute units may not be used in some cycles

Data hazards and stalls may prevent the use of some units
Lack of proper instruction type may also cause idle units



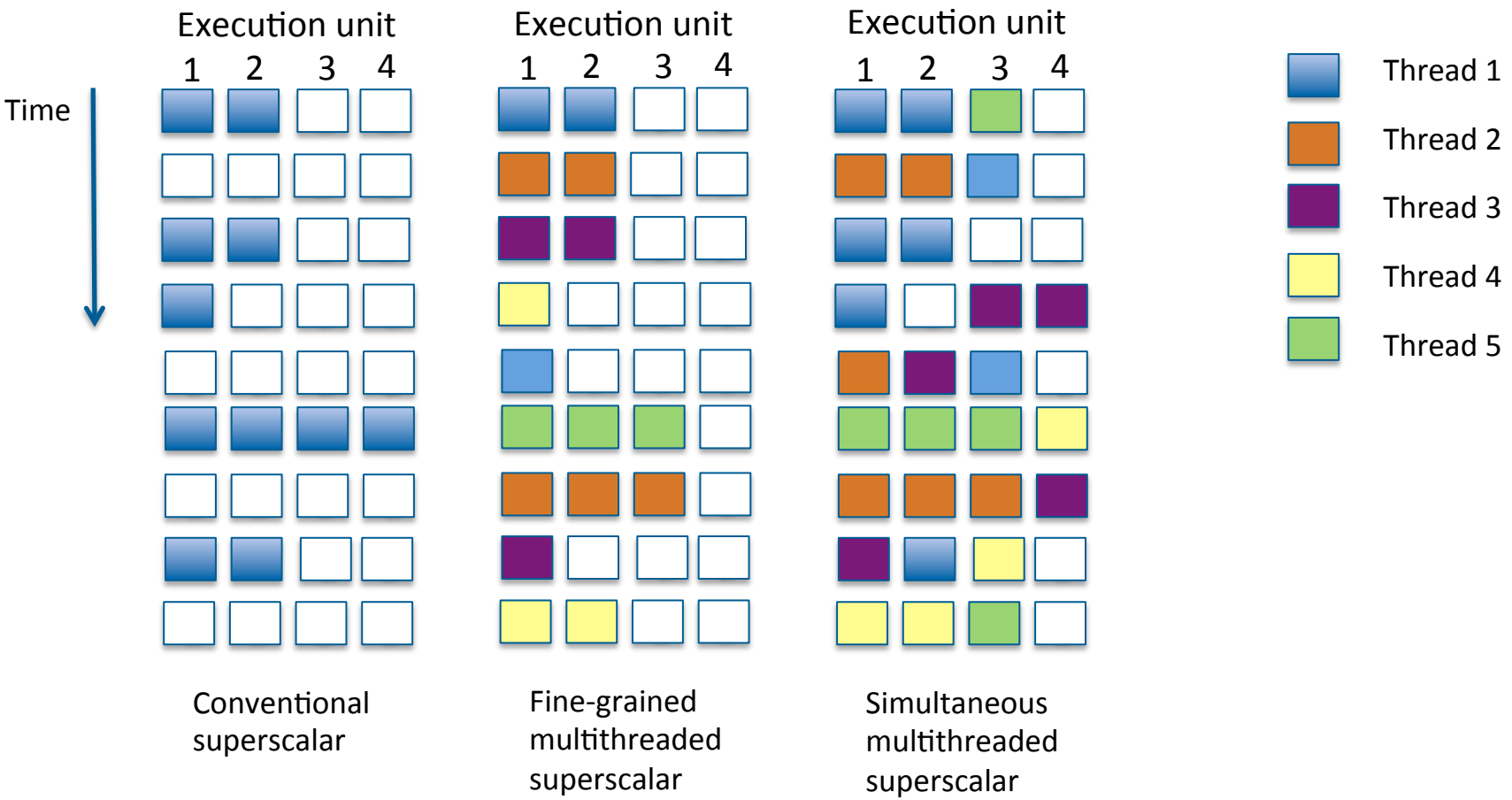
(empty boxes represent idle units)

For example, with 2 integer units and 1 floating point unit, a bubble occurs if there is no floating point instruction available (cycle 2)

The need to stall may idle all 3 units (cycle 8)

SMT requires that each thread have its own register set and PC
Each thread must have its own resources
Instructions are tagged with the thread number or thread ID

Thread Level Parallelism



Simultaneous Multithreading

Hyperthreading (HT) is another name for SMT

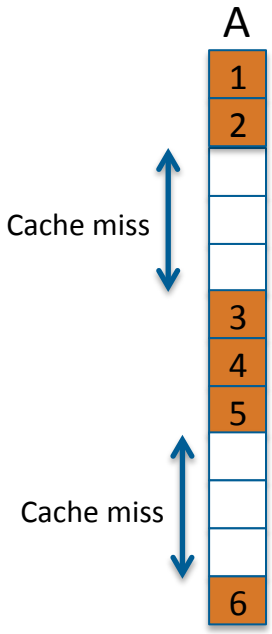
- Intel first used HT in its Xeon processor
- HT is also used in the Pentium 4 & Core i7

- HT provides two logical processors for each physical processor
- Increased Xeon's performance by 30%
- The cost in increased chip area was about 5%

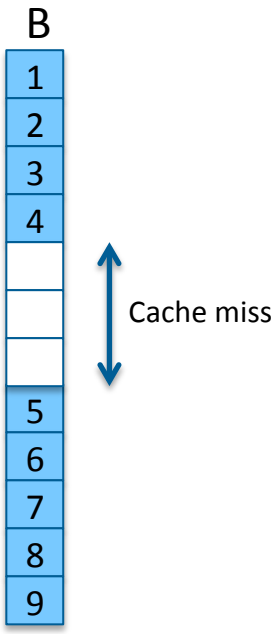
- Requires reproducing registers:
  - Architectural, machine state and control registers

- All other resources are shared by logical processors including:
  - caches, execute units, branch predictors, control logic & buses

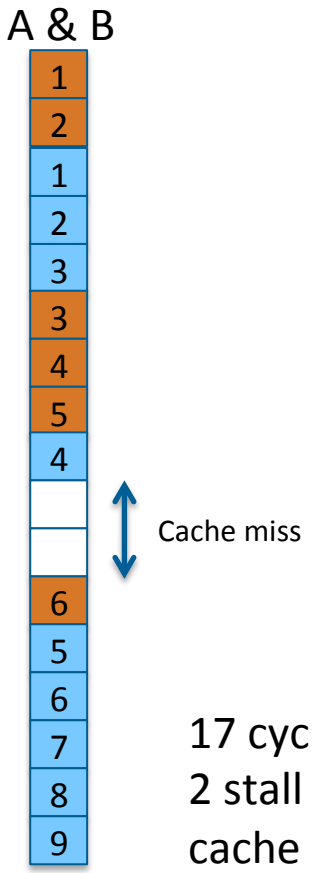
Throughput and Latency



12 cycles total
6 stall cycles due to cache misses
Efficiency = 6/12



12 cycles total
3 stall cycles due to cache misses
Efficiency = 9/12



17 cycles total
2 stall cycles due to cache misses
Efficiency = 15/17
Latency for B increases
Now 17 rather than 12

Motivation for Multi-processing Systems

- Performance can be improved in different ways
  - Higher clock rates
  - Power consumption varies as the square of clock rate
  - Superscalar operation with multiple execute units
  - Requires more complex control units
    - To schedule instructions
    - To avoid dependencies
  - Uses many more transistors
  - Consumes more power and dissipates more heat
  - The degree of improvement reaches a limit

Motivation for Multi-processing Systems

- Multiple less sophisticated processors can be used
  - Provide similar performance at a lower cost
  - Allows slower clock rates
  - Consumes less power
  - Generates less heat
- Coprocessors provide a form of multiprocessing
  - Asymmetric in nature
    - MIPS examples:
      - CP1 for floating point
      - CP0 for managing exceptions
- Vector & array coprocessors may be used

Multi-processor Systems

- Multi-processing Systems
  - Contain more than one processor
  - The processors are identical in an SMP (symmetric multi-processor)
  - Also known as homogeneous

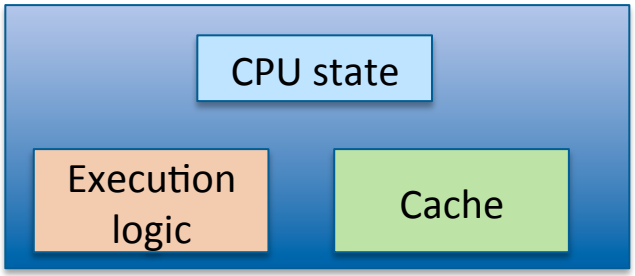
- Each processor plugs into a different socket
  - These are board level multiprocessors
  - A bus connects the processors
  - Provides additional performance on a single computer system
  - To benefit, software must take advantage of the additional processors
  - Inclusion of more processors in an SMP is a choice

Multi-core Systems

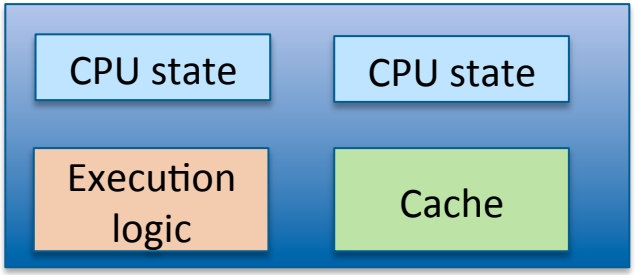
- "Core" logic is replicated on the same chip
  - Multiple cores are presented to the OS
  - Cores share data through on-chip logic or shared caches
  - Homogenous systems use identical cores
  - Heterogeneous systems mix cores having different:
    - ISA
    - Chip area
    - Performance
    - Power dissipation
  - These are chip level multiprocessors
    - Each core has a separate set of registers
    - The number of cores is predetermined
  - Issues relating to shared memory must be handled

Multi-core Systems

- Multi-core concepts & designs developed over time



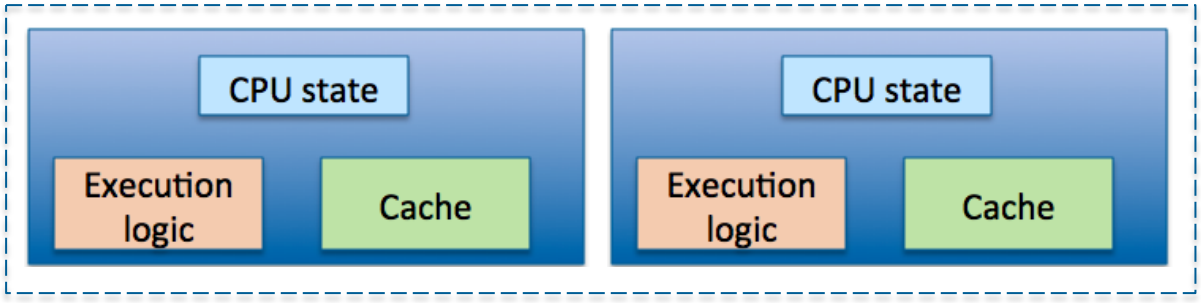
Basic single-core processor



Single-core processor with hyperthreading

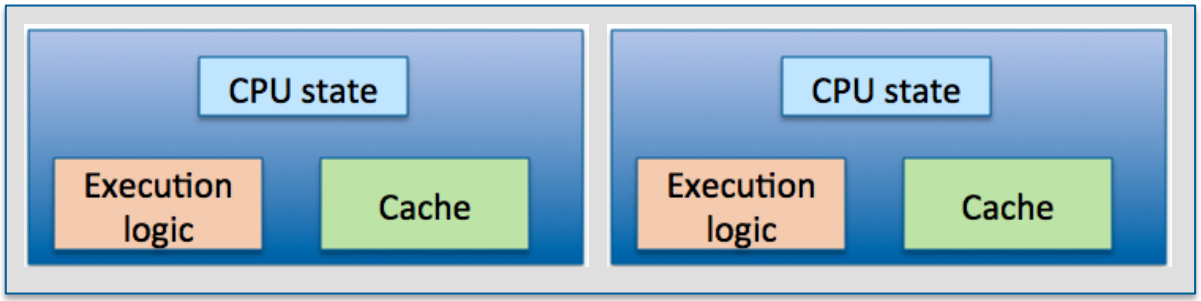
Hyperthreading also known as simultaneous multithreading uses additional hardware registers to allow one physical processor to act as two virtual processors

Multi-core Systems



Two or more separate processors

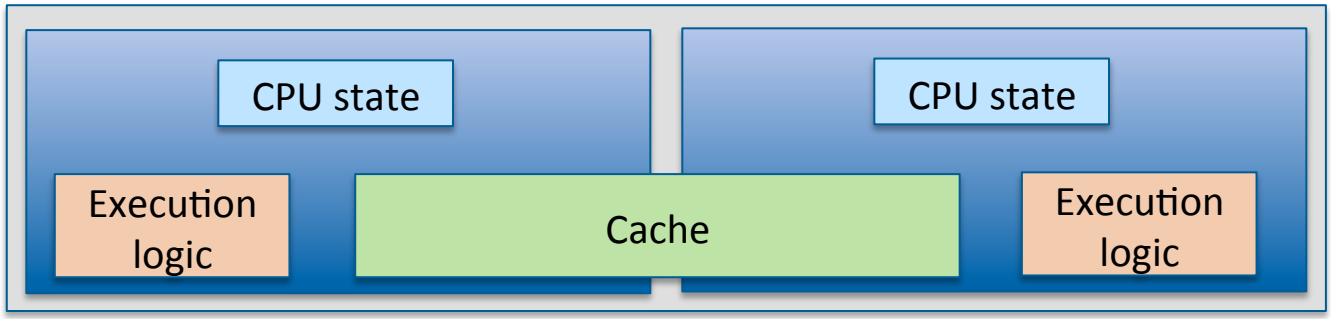
The processors share memory and the same environment (motherboard)
The dashed lines mean they are in the same system but not on the same chip



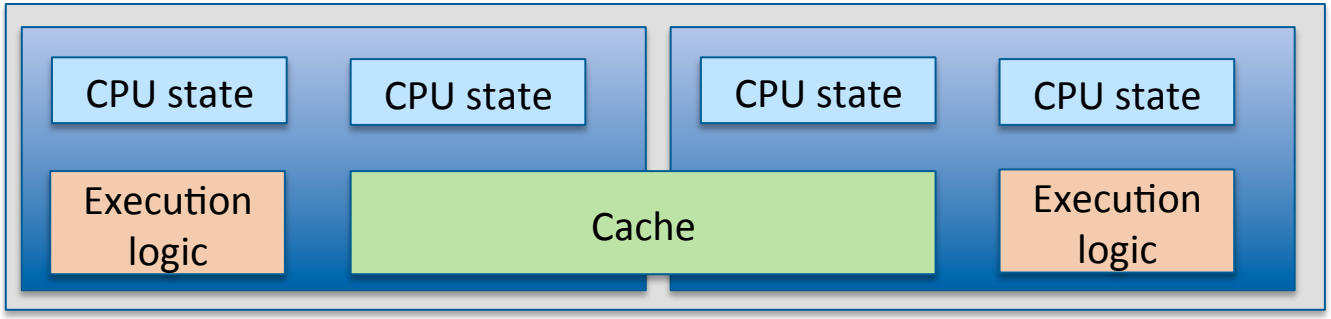
Multiple cores

Multi-core processor with two or more cores housed in the same package.

Multi-core Systems



Cores share a common cache

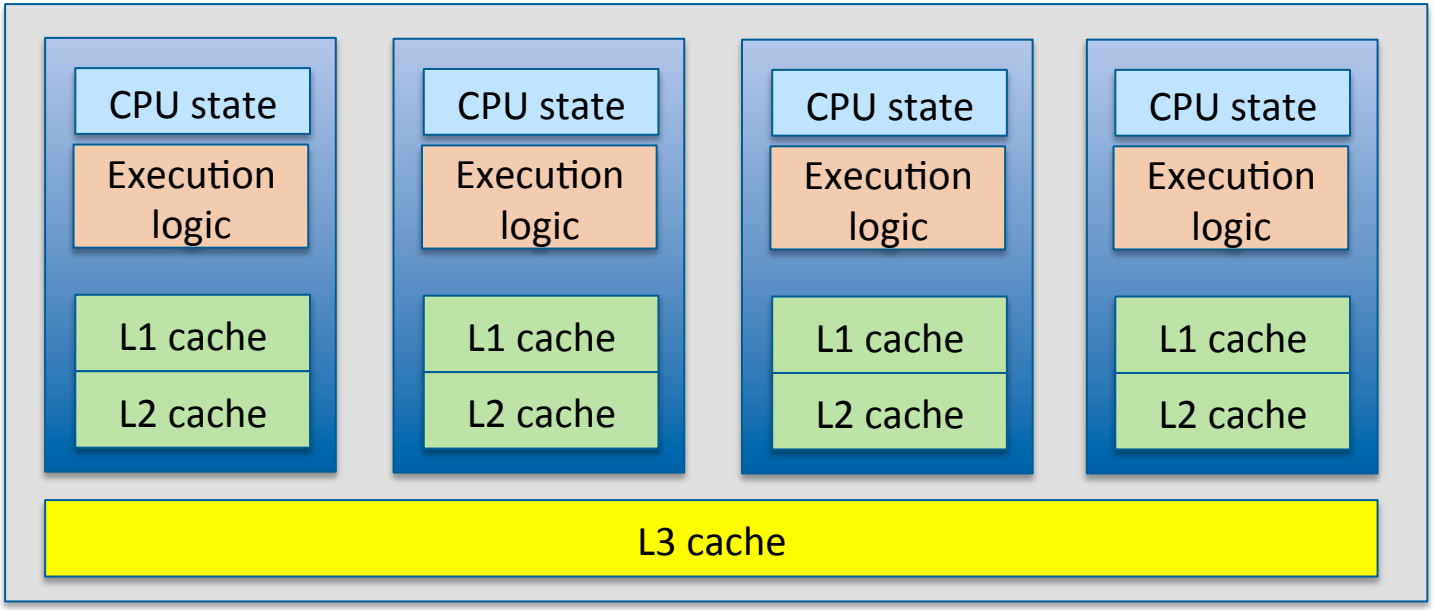


Dual-core processor with hyperthreaded cores

Sharing the cache increases the degree of coupling between the processor cores

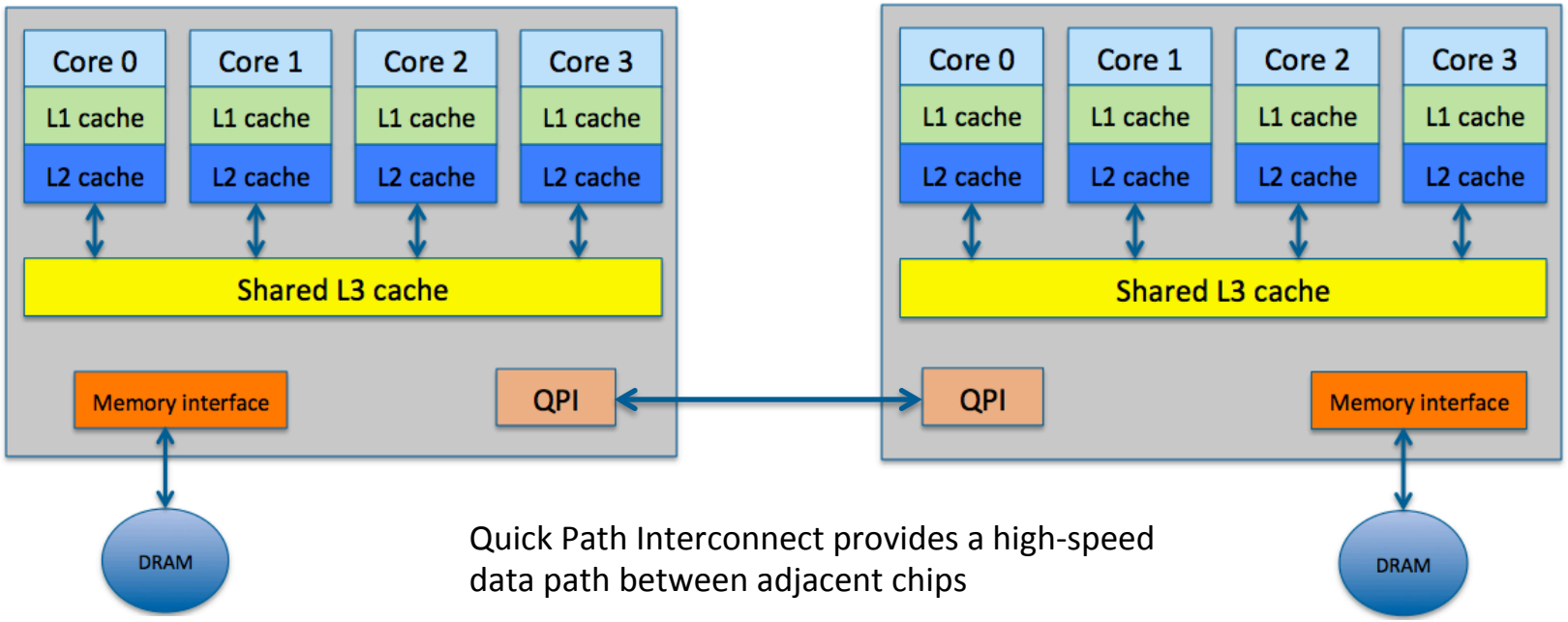
Multi-core Systems

Quad-core system



Each processor has a private L1 and L2 cache
All processors share a much larger L3 cache

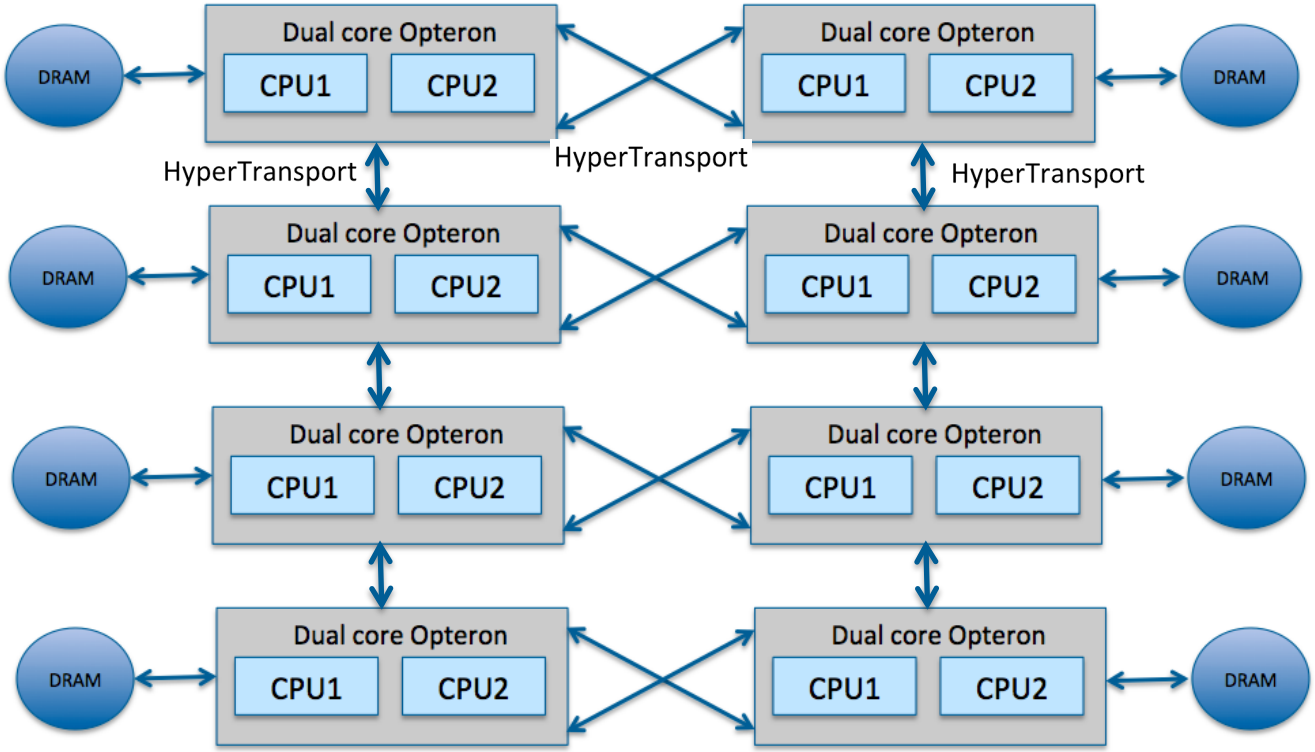
Intel Nehalem Multi-Core Processor



Quick Path Interconnect provides a high-speed data path between adjacent chips

- Introduced in 2008 with the Core i7
  - The QPI interface transfers data on both the rising and falling edge of the clock (at 3.2 GHz) for a bandwidth of 25.6GB/s.

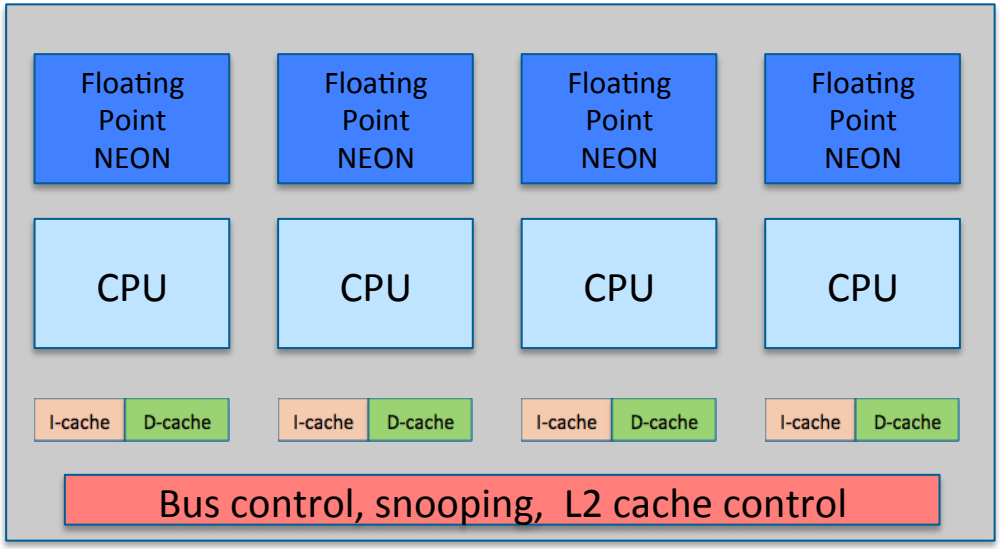
AMDs Multi-core System



Has 8 dual-core processors linked via HyperTransport

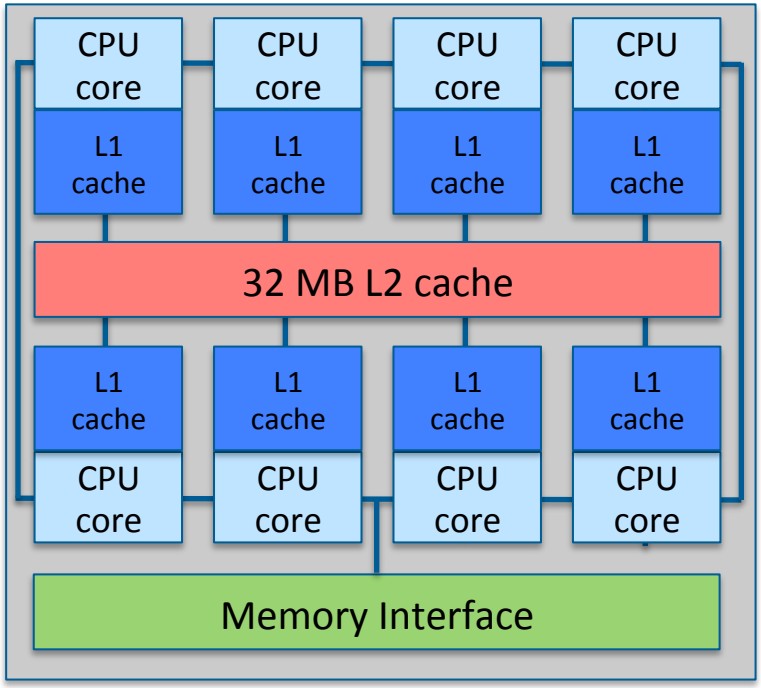
- A NUMA system with a high speed cross-linked network

ARM's Cortex A9 multi-core system



A general-purpose processor intended for use in mobile devices
Available with 1, 2 , 3, or 4 superscalar cores
Can directly execute Java bytecode
No L2 cache, but includes on chip bus snooping control

IBM Power7



Eight cores, can be turned off individually to save energy
Cores contain 2 integer ALUs & 2 floating point units
Also includes a decimal floating-point (DFP) unit for financial applications
Hardware DFP is 100 to 1000 times faster than software decimal arithmetic