



International
Centre for
Radio
Astronomy
Research

Parallel Computing

Research Associate Professor
Kevin Vinsen



The problem

Imagine a program that processes requests and has a global count

```
totalRequests = totalRequests + 1
```

The compiler might produce the following:

```
// load memory for totalRequests into register
```

```
MOV EAX, [totalRequests]
```

```
// update register
```

```
INC EAX
```

```
// store updated value back to memory
```

```
MOV [totalRequests], EAX
```



Races in detail

What might happen if two threads run this at the same time?

Thread A

```
→ MOV EAX,[totalRequests]
EAX = ? Inc EAX
MOV [totalRequests],EAX
```

```
MOV EAX,[totalRequests]
EAX = 8 → Inc EAX
MOV [totalRequests],EAX
```

```
MOV EAX,[totalRequests]
EAX = 9 Inc EAX
→ MOV [totalRequests],EAX
```

```
MOV EAX,[totalRequests]
EAX = 9 Inc EAX
MOV [totalRequests],EAX
```

```
MOV EAX,[totalRequests]
EAX = 9 Inc EAX
MOV [totalRequests],EAX
```

Memory

```
totalRequests = 8
time = 0
```

```
totalRequests = 8
time = 1
```

```
totalRequests = 9
time = 2
```

```
totalRequests = 9
time = 3
```

```
totalRequests = 9
time = 4
```

Thread B

```
MOV EAX,[totalRequests]
EAX = ? Inc EAX
MOV [totalRequests],EAX
```

```
→ MOV EAX,[totalRequests]
EAX = ? Inc EAX
MOV [totalRequests],EAX
```

```
MOV EAX,[totalRequests]
EAX = 8 → Inc EAX
MOV [totalRequests],EAX
```

```
MOV EAX,[totalRequests]
EAX = 9 Inc EAX
→ MOV [totalRequests],EAX
```

```
MOV EAX,[totalRequests]
EAX = 9 Inc EAX
MOV [totalRequests],EAX
```



Races in details

Oh @\$\$\$#*! - Wrong answer

totalRequest is 9 not 10



The basic idea

Spread operations over many processors

If n operations take time t on 1 processor,

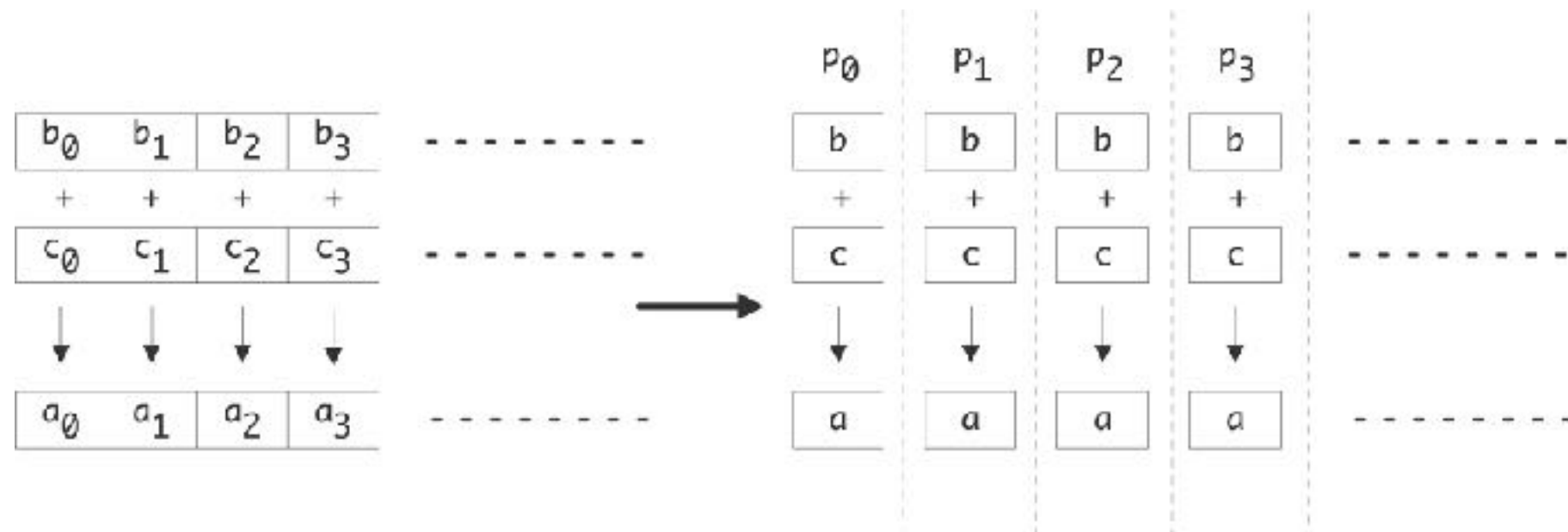
Does this become t/p on p processors ($p \leq n$)?

```
for (i=0; i<n; i++)  
    a[i] = b[i]+c[i]
```

```
a = b+c
```

Idealised version: every process
has one array element

The basic idea





The basic idea

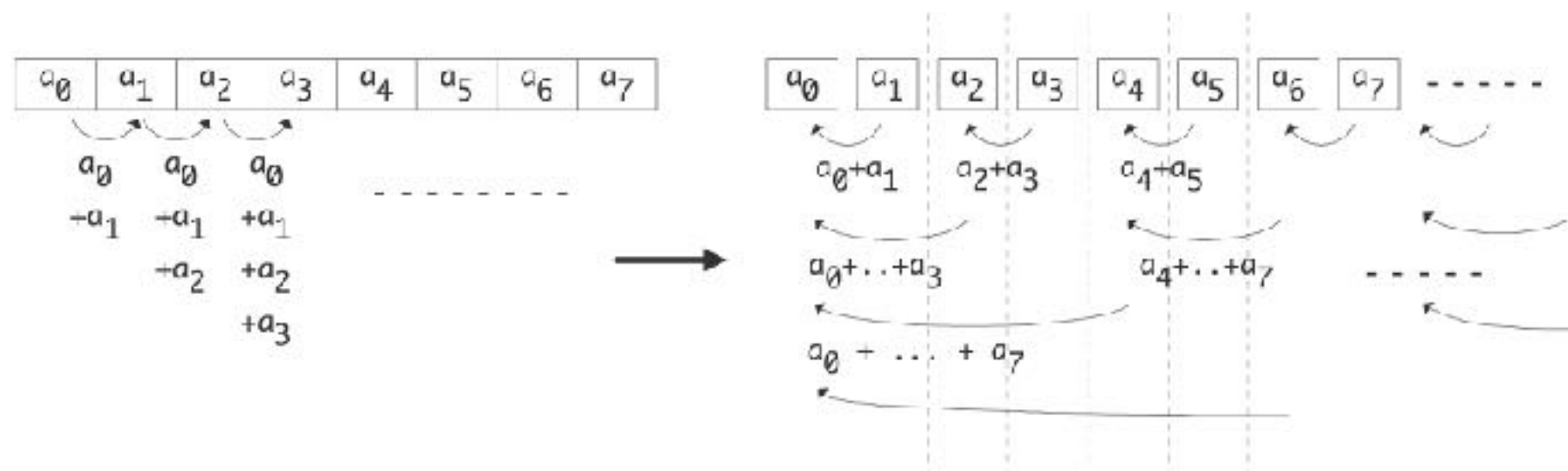
Spread operations over many processors

If n operations take time t on 1 processor,

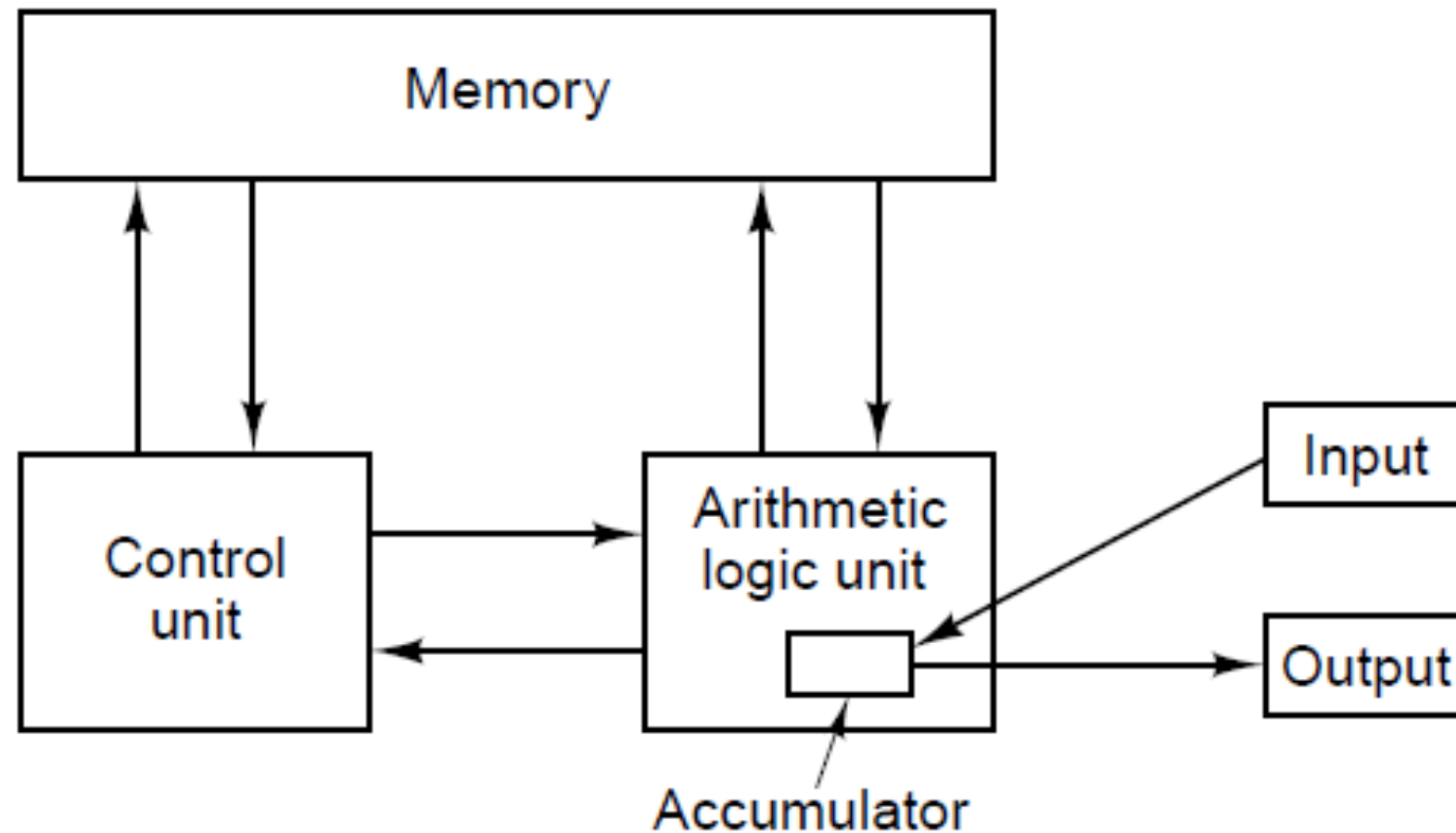
Does it always become t/p on p processors ($p \leq n$)?

```
s = sum( x[i], i=0,n-1 )
```

The basic idea



von Neumann Machine



The original von Neumann machine.



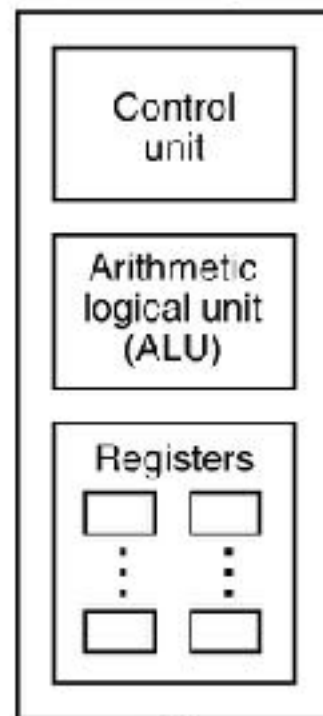
Von Neumann Architecture

- **Instruction decode: determine operation and operands**
- **Get operands from memory**
- **Perform operation**
- **Write results back**
- **Continue with next instruction**

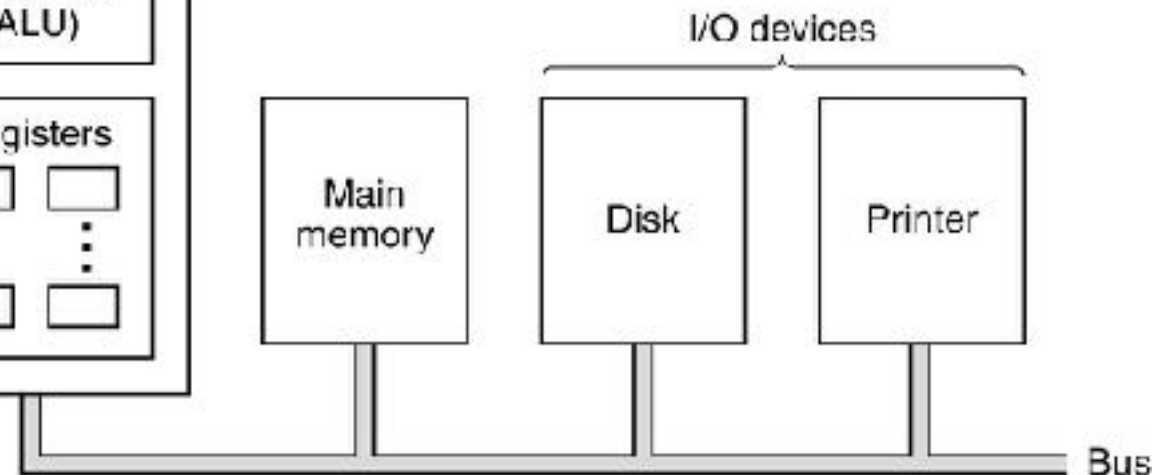


Simple Architecture

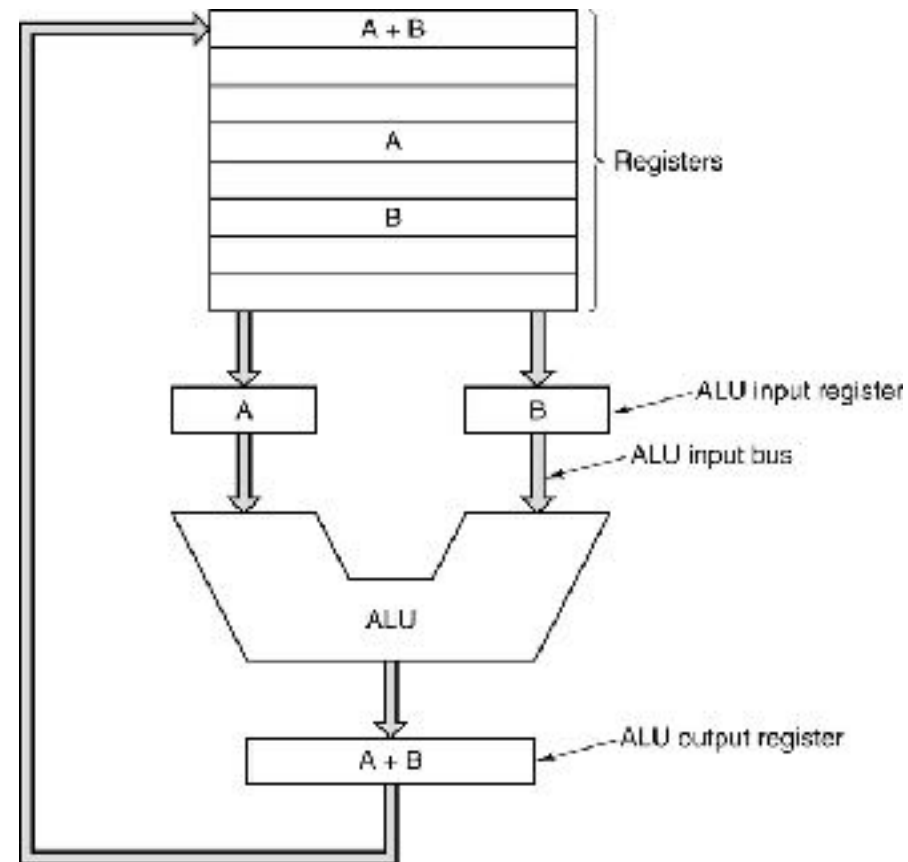
Central processing unit (CPU)



The organisation of a simple computer with one CPU and two I/O devices



CPU Organisation



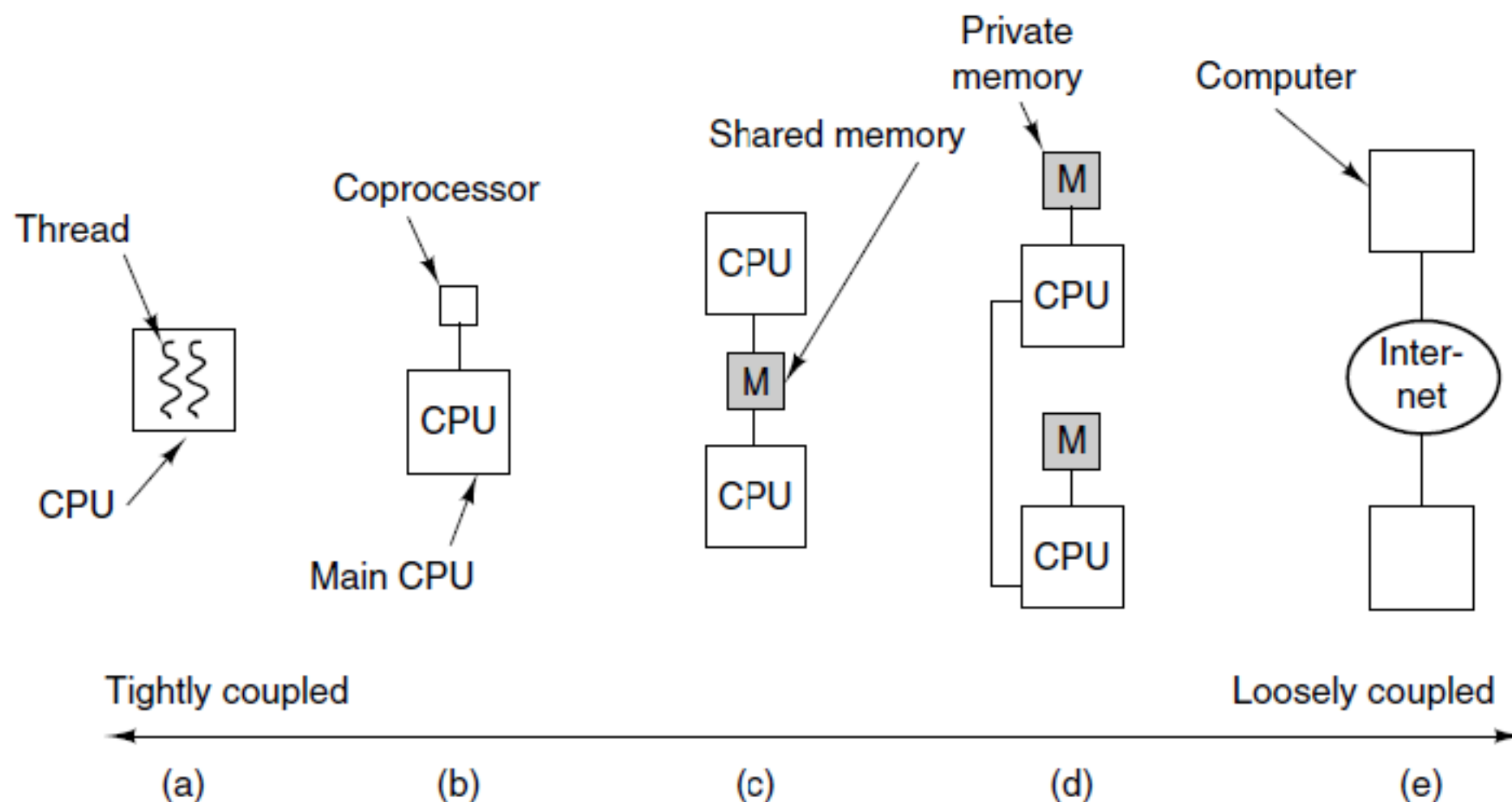
The data path of a typical Von Neumann machine.



Instruction Execution Steps

1. Fetch next instruction from memory into instr. register
2. Change program counter to point to next instruction
3. Determine type of instruction just fetched
4. If instructions uses word in memory, determine where
Fetch word, if needed, into CPU register
5. Execute the instruction
6. Go to step 1 to begin executing following instruction

Parallelism options



(a) On-chip parallelism. (b) A coprocessor. (c) A multiprocessor.
(d) A multicomputer. (e) A grid.

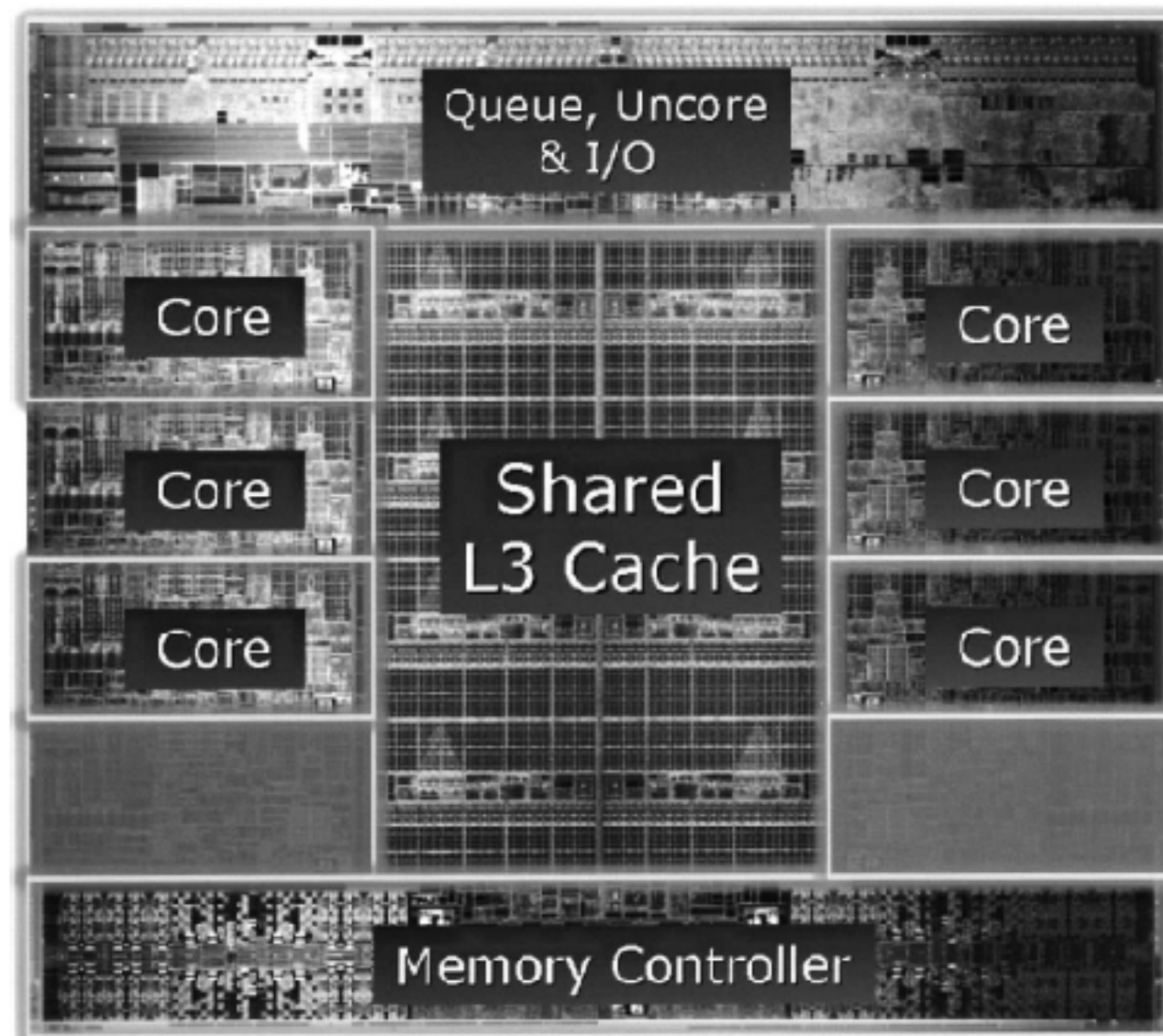


Lots of tricks to speed things up

- Multiple operations simultaneously “in flight”
- Operands can be in memory, cache, register
- Results may need to be coordinated with other processing elements
- Operations can be performed speculatively



The Intel Core i7-3960X die.

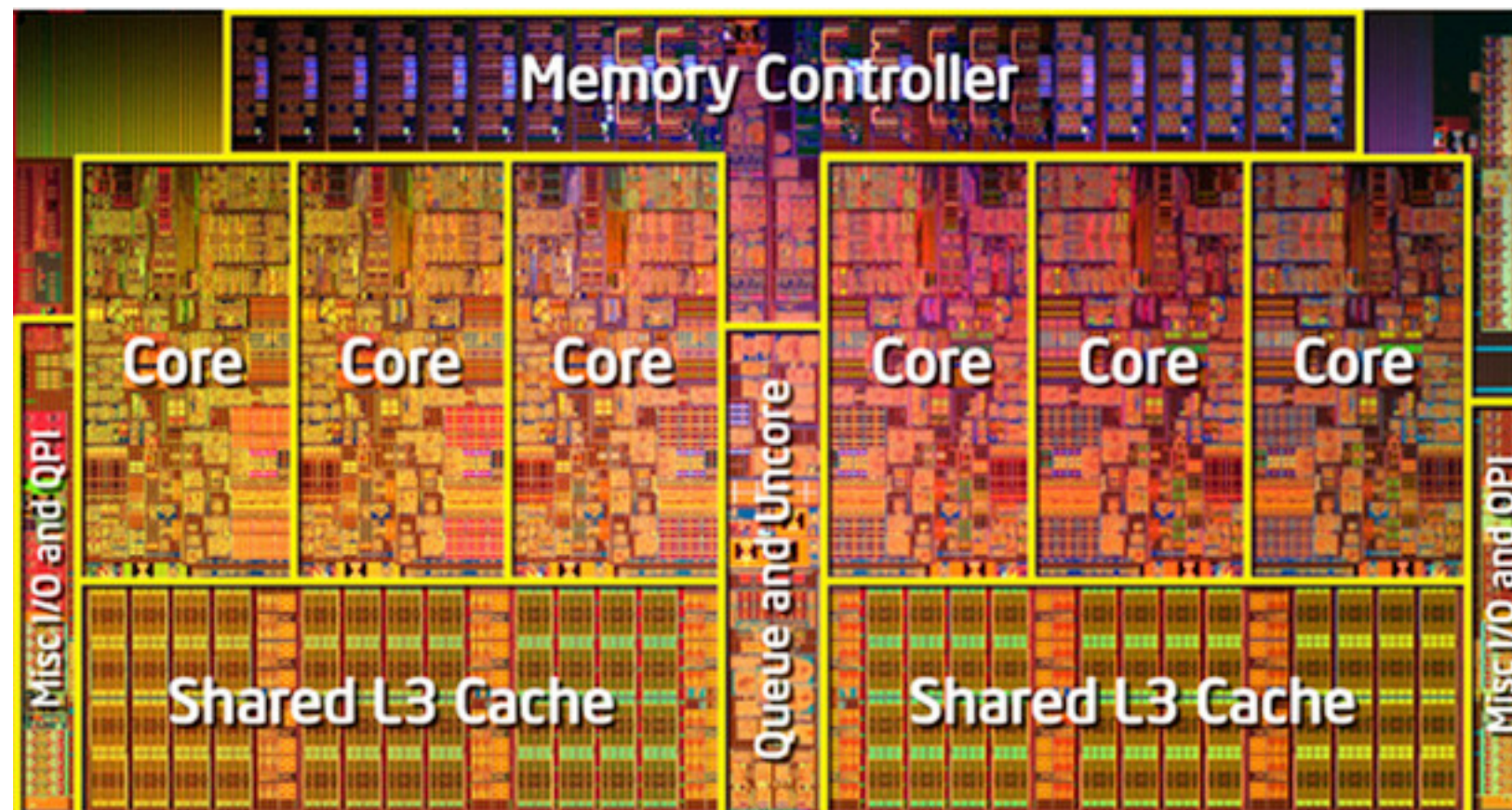


The die is 21 by 21 mm and has 2.27 billion transistors.

© 2011 Intel Corporation.

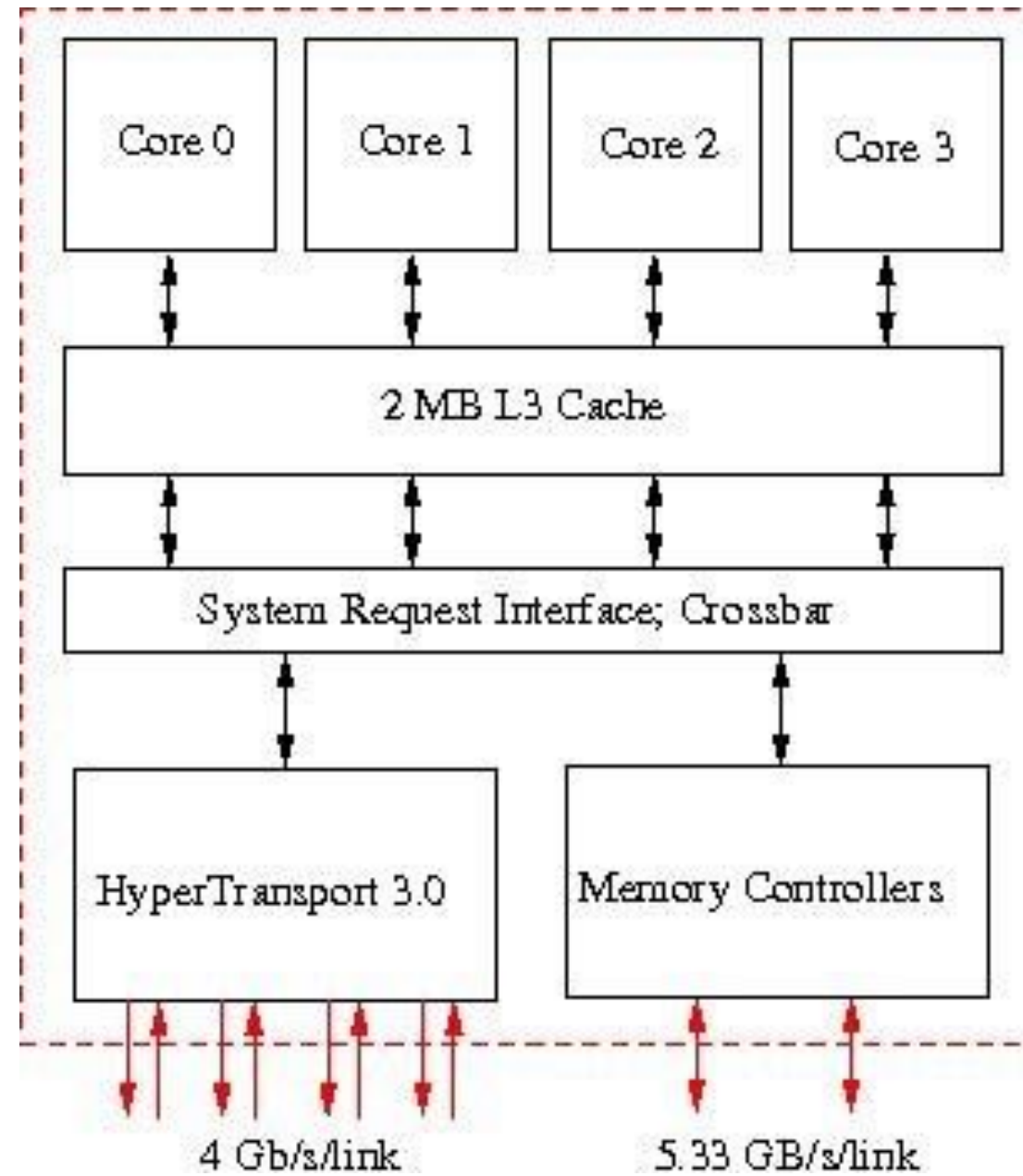


The Intel Core i7-970 die.



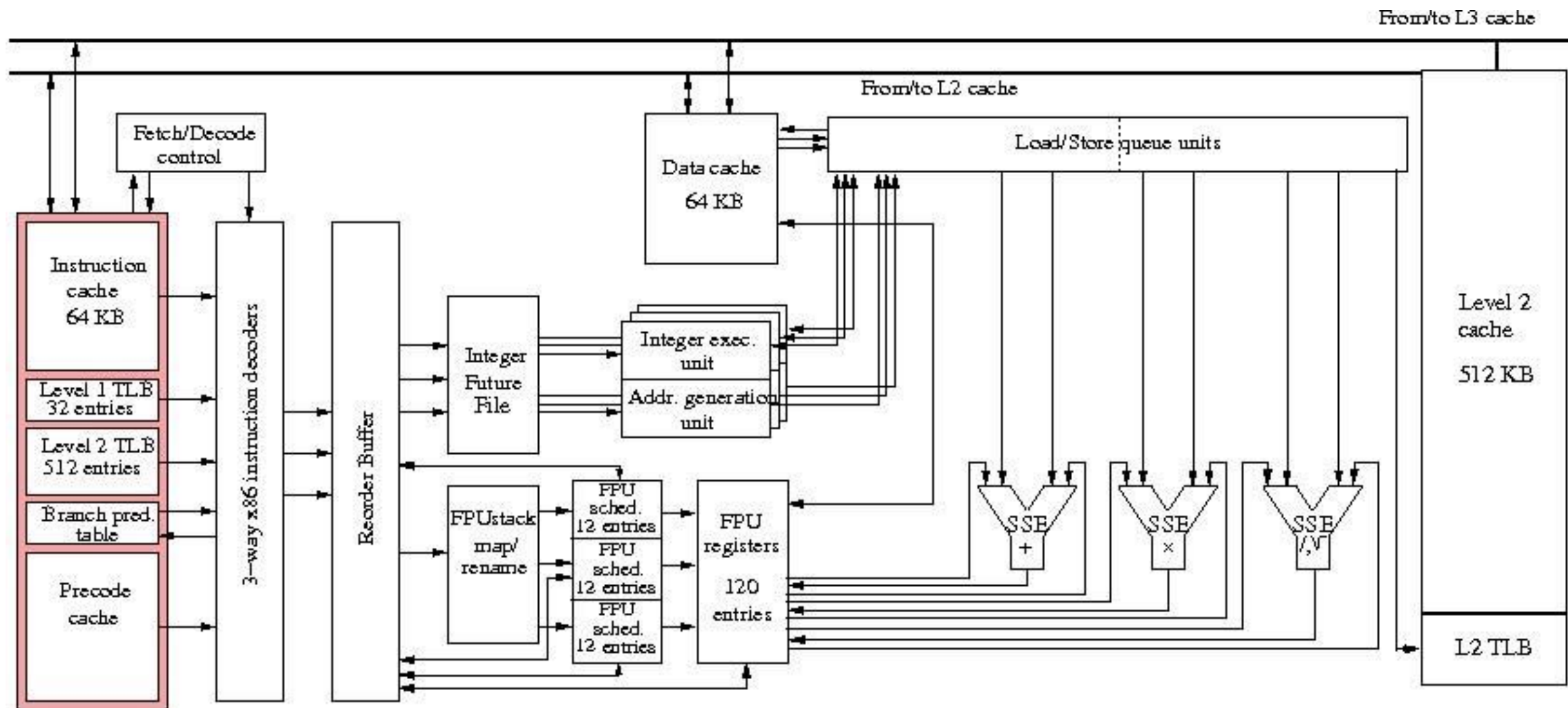


Logical diagram





What is in a core





Design Principles for Modern Computers

All instructions directly executed by hardware
Maximise rate at which instructions are issued
Instructions should be easy to decode
Only loads, stores should reference memory
Provide plenty of registers



Pipelining

A single instruction takes several clock cycles to complete

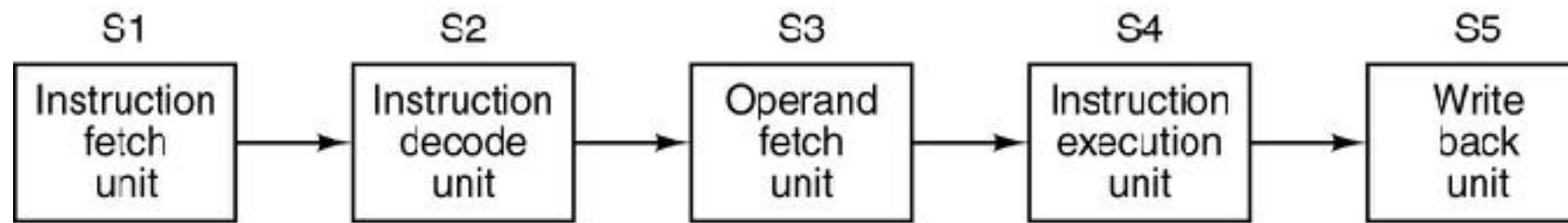
Subdivide an instruction:

- Instruction decode
- Operand exponent align
- Actual operation
- Normalize

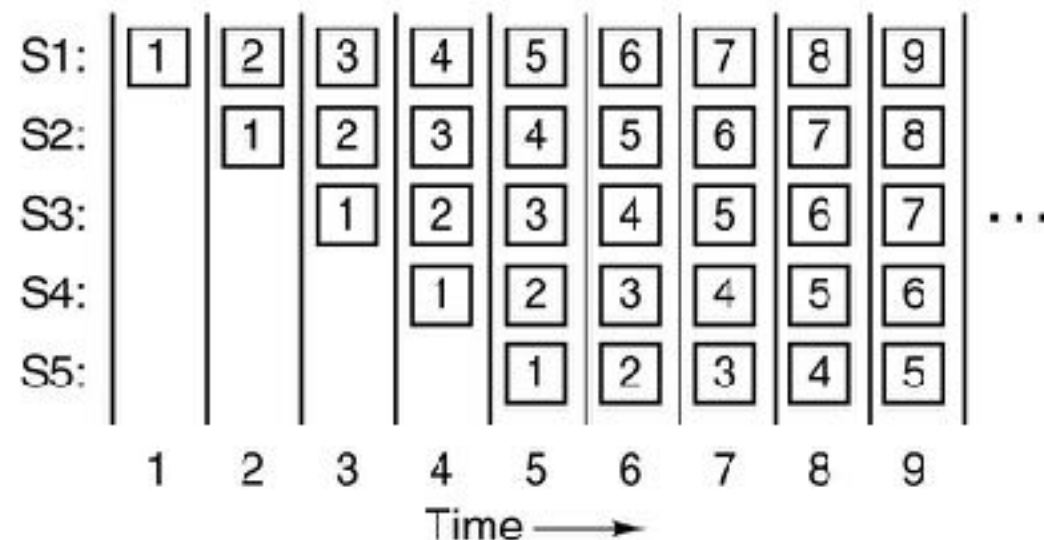
Pipeline: separate piece of hardware for each subdivision
Compare to assembly line



Instruction-Level Parallelism



(a)



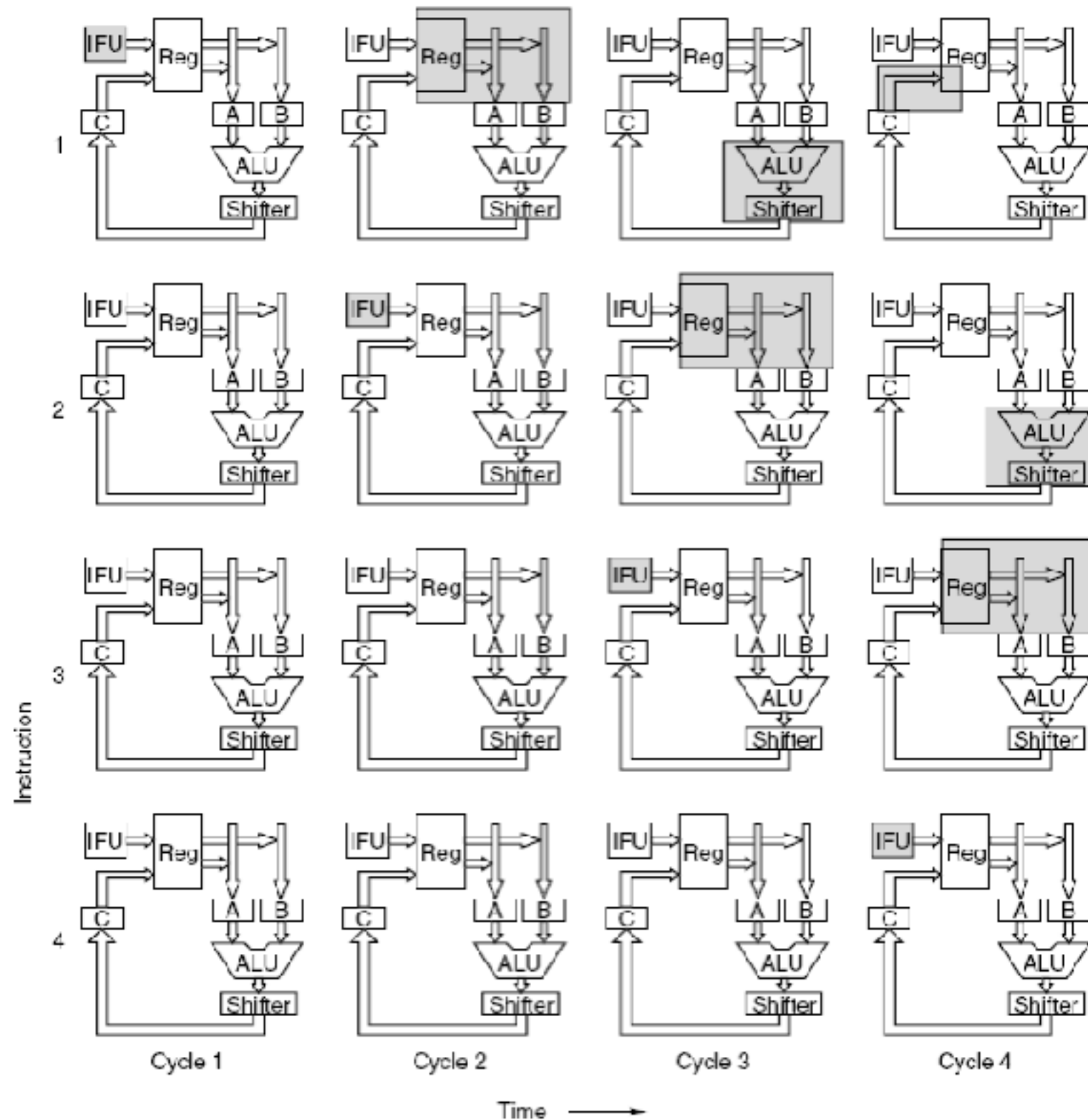
(b)

(a) A five-stage pipeline.

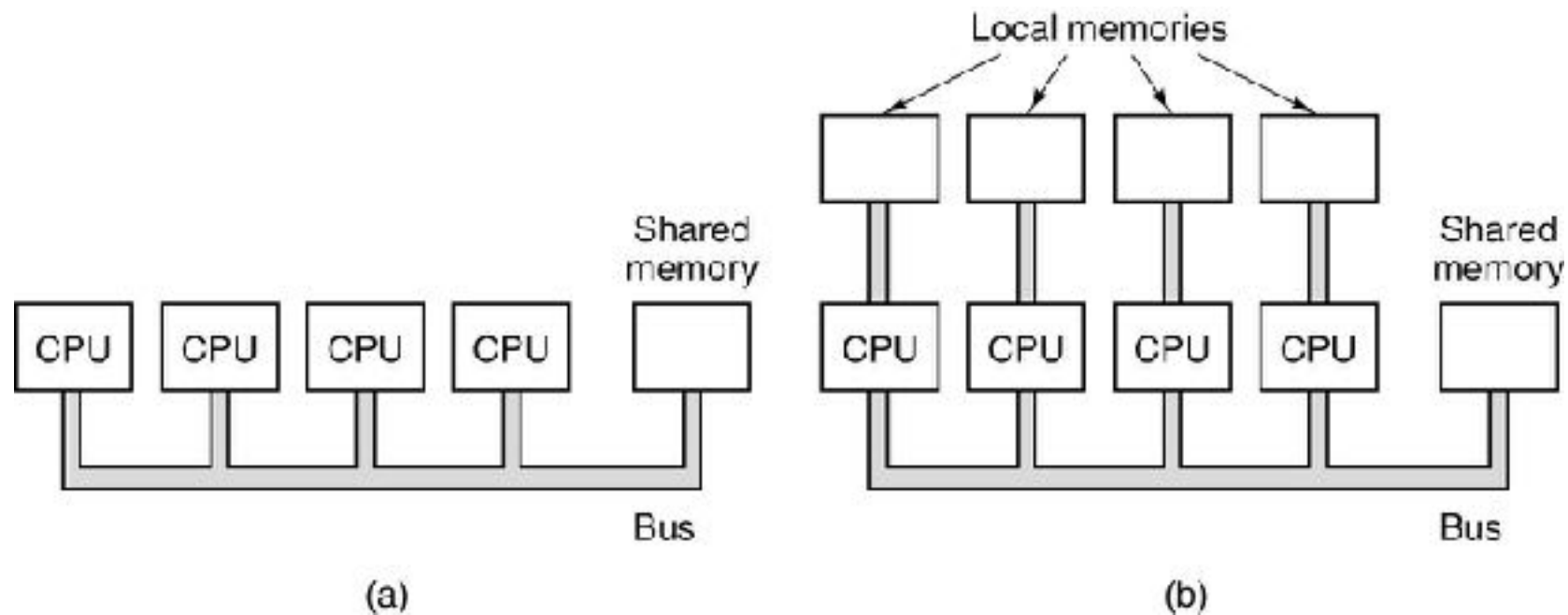
(b) The state of each stage as a function of time. Nine clock cycles are illustrated

Pipeline

Graphical illustration of how a pipeline works.



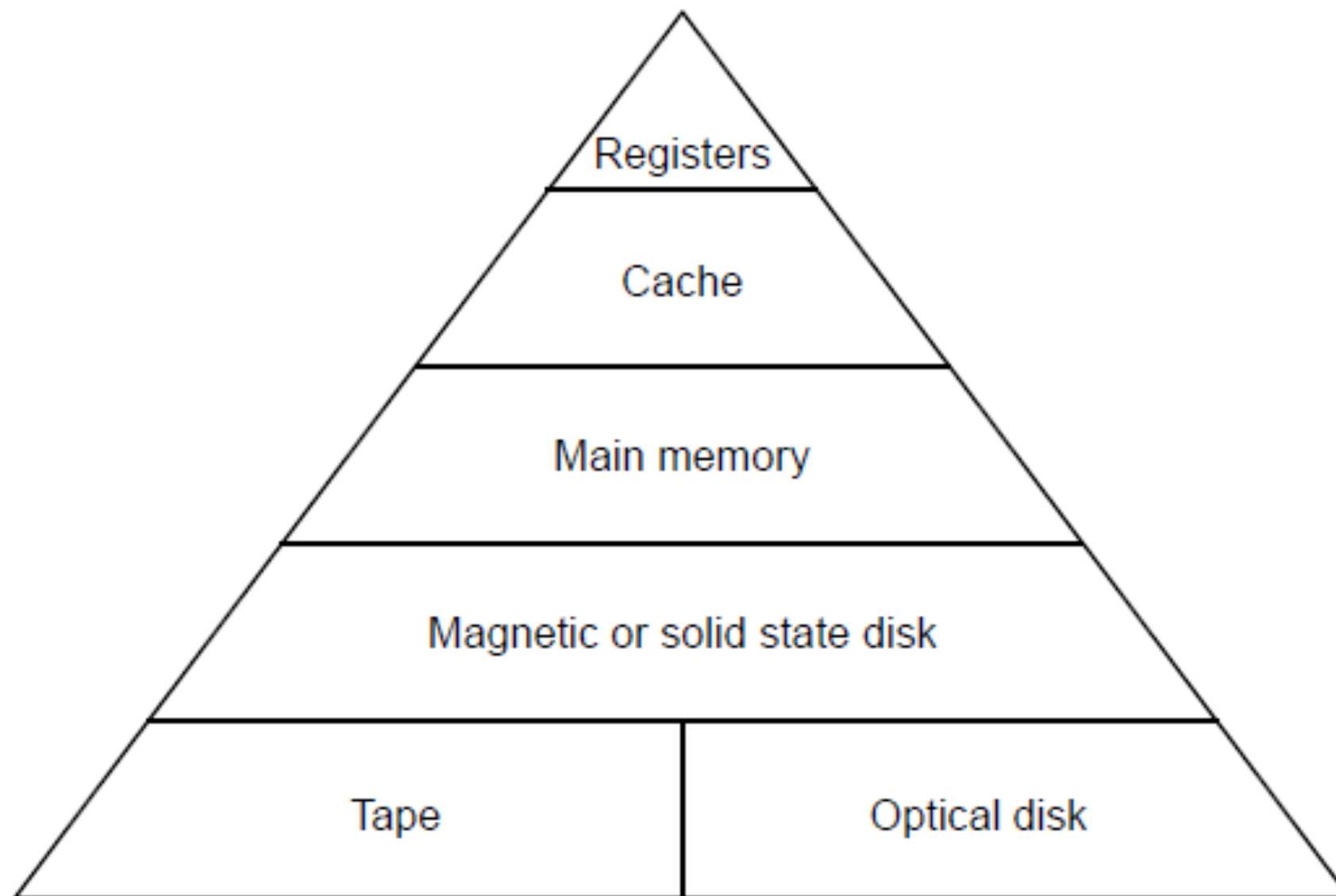
Processor level parallelism



- a) A single-bus multiprocessor.
- b) A multicomputer with local memories.



Secondary Memory Hierarchies



A five-level memory hierarchy.



Memory Hierarchies

Memory is divided into different levels:

Registers

Caches

Main Memory

Memory is accessed through the hierarchy

registers where possible

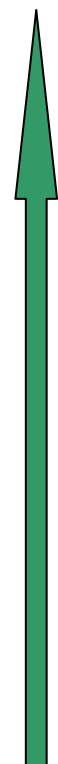
... then the caches

... then main memory



Memory Relativity

SPEED



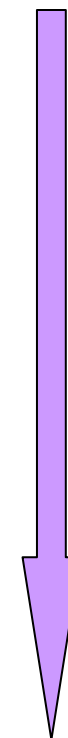
CPU
Registers: 16

L1 cache
(SRAM, 64k)

L2 cache
(SRAM, 1M)

MEMORY
(DRAM, >1G)

SIZE



Cost (\$/bit)





Latency and Bandwidth

The two most important terms related to performance for memory subsystems and for networks are:

Latency

- How long does it take to retrieve a word of memory?
- Units are generally nanoseconds (milliseconds for network latency) or clock periods (CP).
- Sometimes addresses are predictable: compiler will schedule the fetch. Predictable code is good!

Bandwidth

- What data rate can be sustained once the message is started?
- Units are B/sec (MB/sec, GB/sec, etc.)

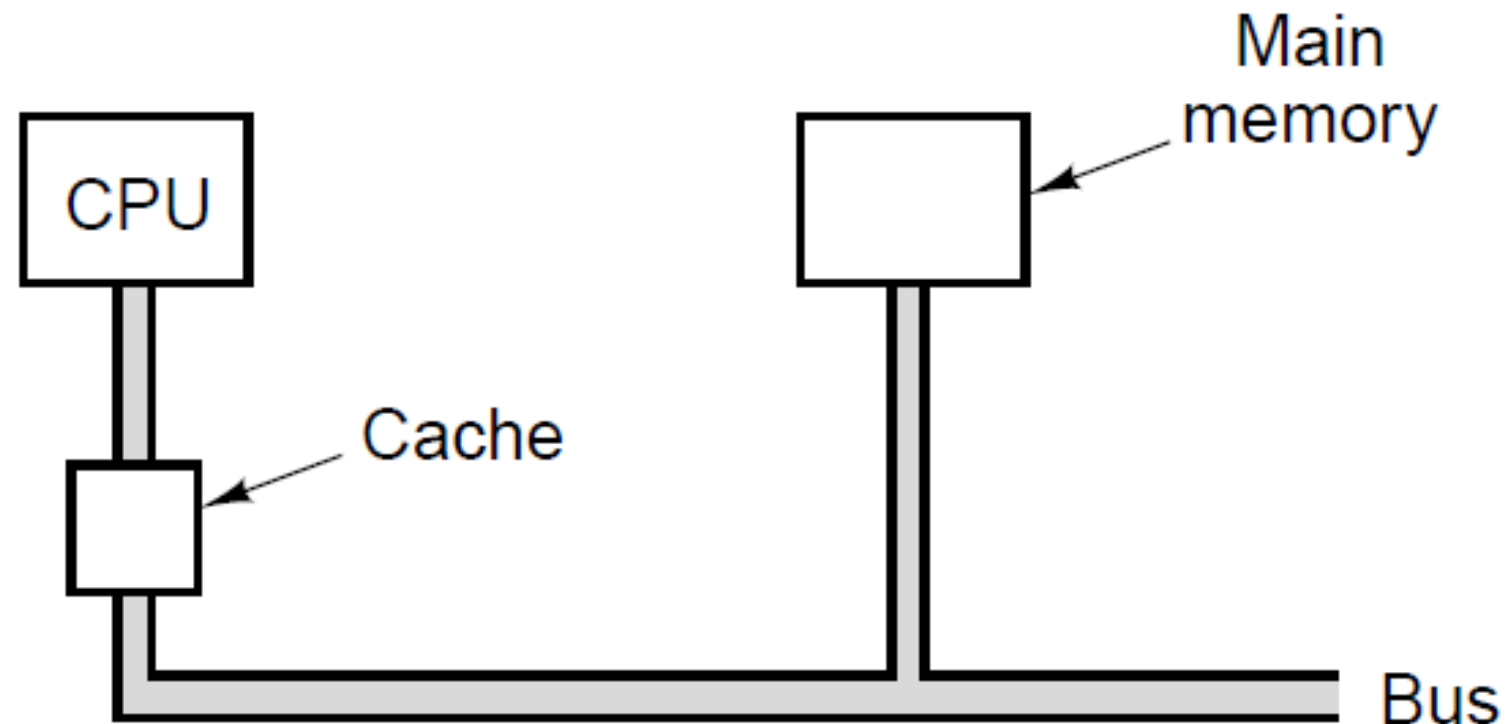


Latency hiding & GPUs

- Finding parallelism is sometimes called ‘latency hiding’: load data early to hide latency
- GPUs do latency hiding by spawning many threads (recall CUDA SIMD programming): SIMT
- Requires fast context switch



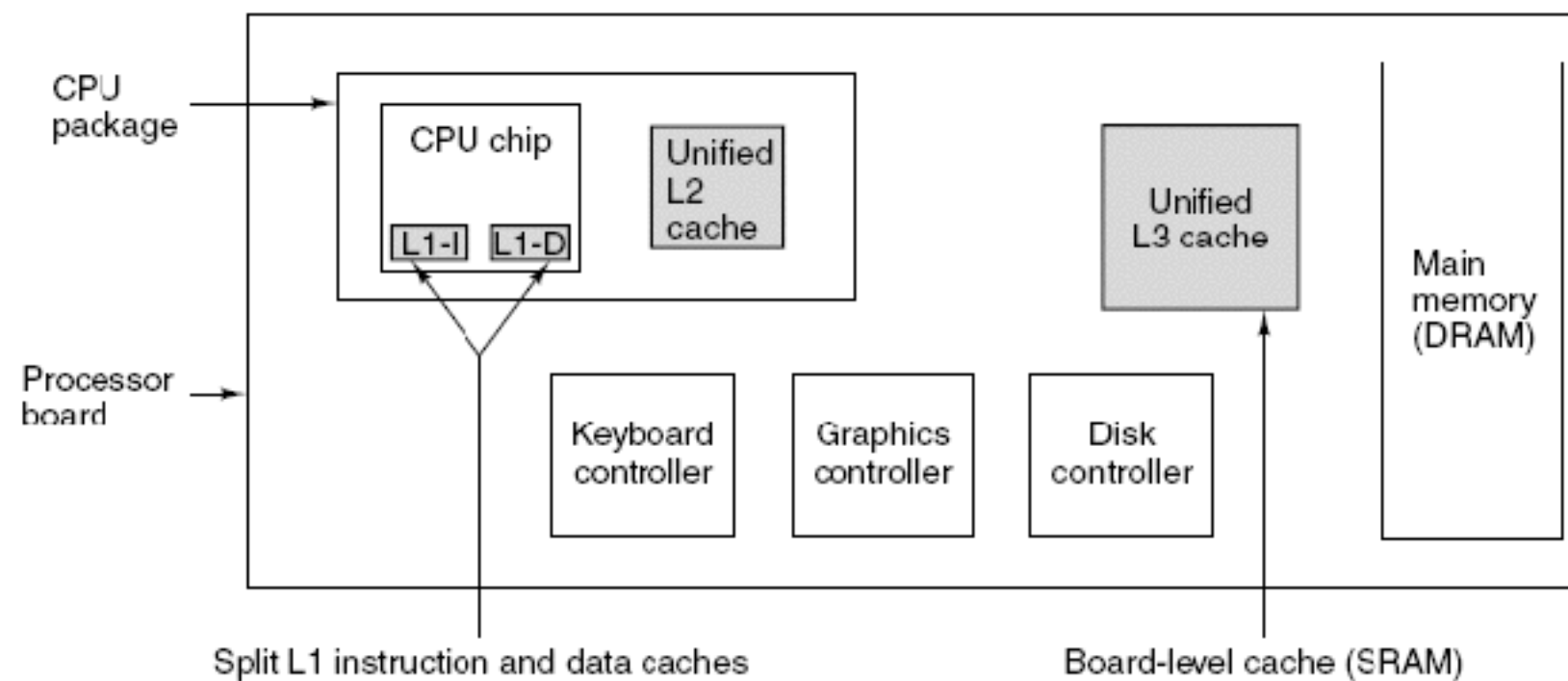
Cache Memory



The cache is logically between the CPU and main memory. Physically, there are several possible places it could be located.



Cache Memory



A system with three levels of cache.



Branch Prediction

```
if (i == 0)
    k = 1;
else
    k = 2;
```

(a)

```
                CMP i,0    ; compare i to 0
                BNE Else    ; branch to Else if not equal
Then:           MOV k,1     ; move 1 to k
                BR Next     ; unconditional branch to Next
Else:           MOV k,2     ; move 2 to k
Next:
```

(b)

- (a) A program fragment.
(b) Its translation to a generic assembly language.



Static Branch Prediction

```
for (i = 0; i < 1000000; i++) {  
    // Do something
```



Out-of-Order Execution and Register Renaming

					Registers being read								Registers being written							
Cy	#	Decoded	Iss	Ret	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	–		3	2	1									1	1	1		
3					3	2	1									1	1	1		
4				1	2	1	1										1	1		
				2	1	1												1		
				3																
5			4			1			1										1	
	5	R7=R1*R2	5			2	1		1										1	1
6	6	R1=R0–R2	–			2	1		1										1	1
7				4		1	1													1
8				5																

A superscalar CPU with in-order issue and in-order completion.

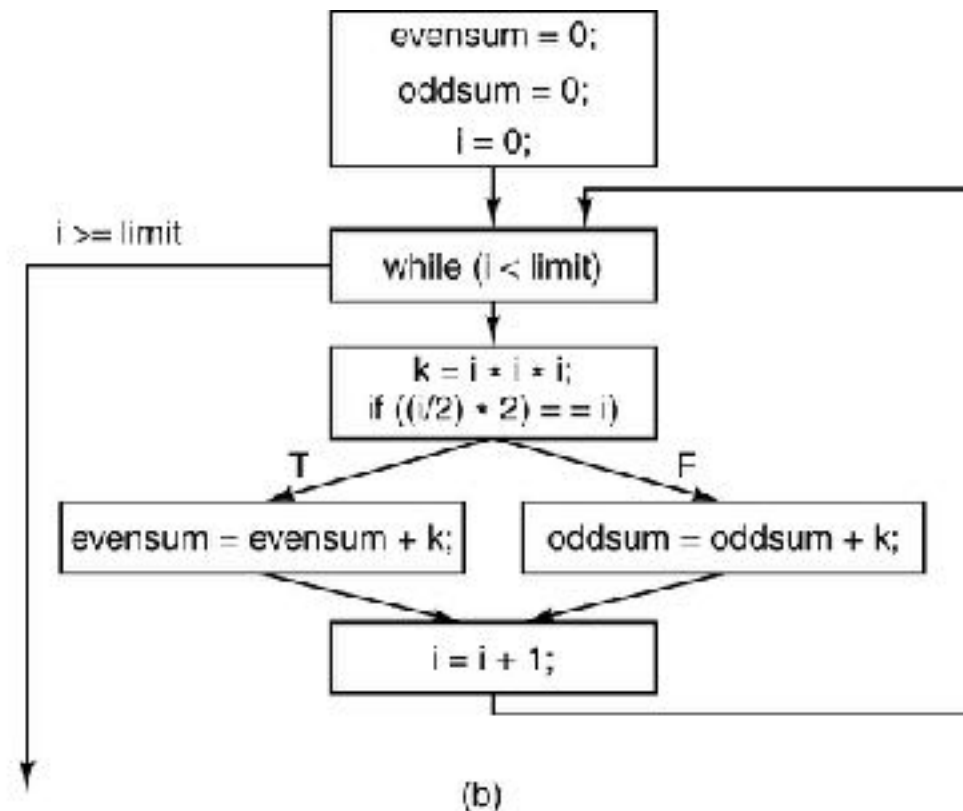
Speculative Execution

```

evensum = 0;
oddsum = 0;
i = 0;
while (i < limit) {
    k = i * i * i;
    if ((i/2) * 2 == i)
        evensum = evensum + k;
    else
        oddsum = oddsum + k;
    i = i + 1;
}
    
```

(a)

A program fragment



(b)

The corresponding basic block graph.



Parallel Computer Architectures

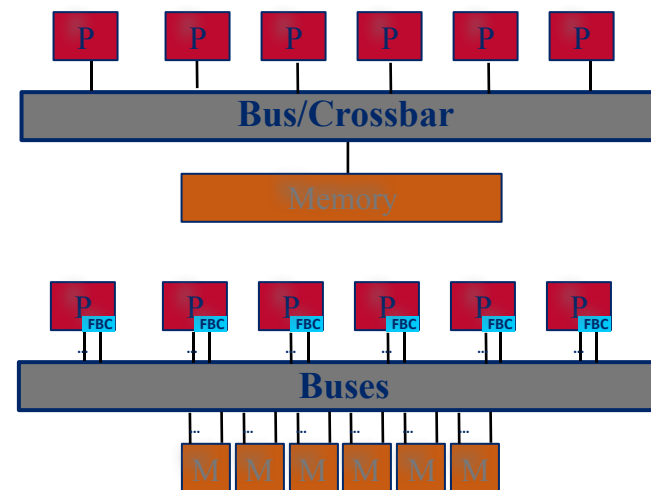
Shared memory: all processors share the same address space

- OpenMP: directives-based programming

Distributed memory: every processor has its own address space

- MPI: Message Passing Interface

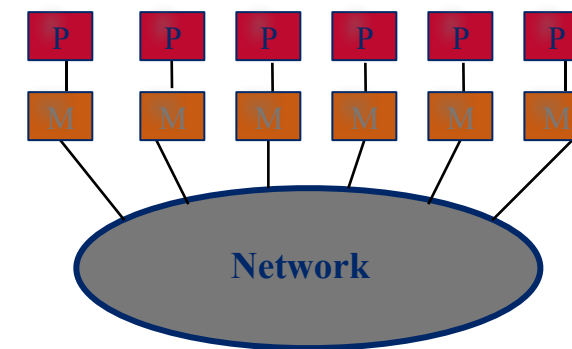
Shared and Distributed Memory



Shared memory: single address space. All processors have access to a pool of shared memory.
(e.g., Single Cluster node (2-way, 4-way, ...))

Methods of memory access :

- Bus
- Distributed Switch
- Crossbar

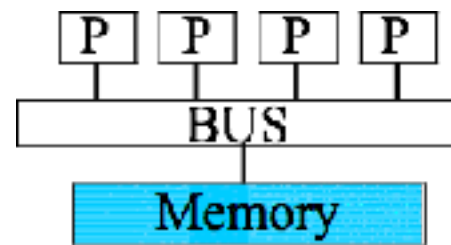


Distributed memory: each processor has its own local memory. Must do message passing to exchange data between processors.
(examples: Linux Clusters, Cray XT3)

Methods of memory access :

- single switch or switch hierarchy with fat tree, etc. topology

Shared Memory: UMA and NUMA

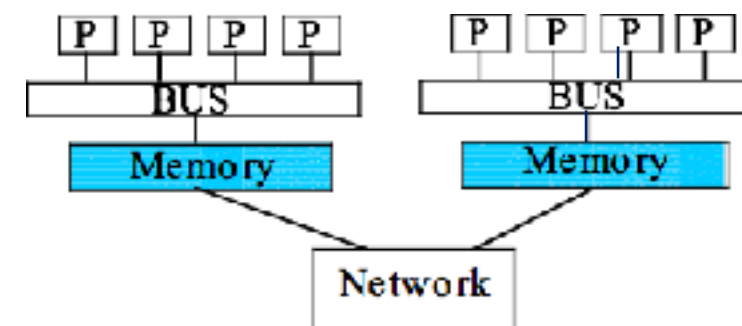


Uniform Memory Access (UMA):

Each processor has uniform access time to memory - also known as symmetric multiprocessors (SMPs) (example: Sun E25000 at TACC)

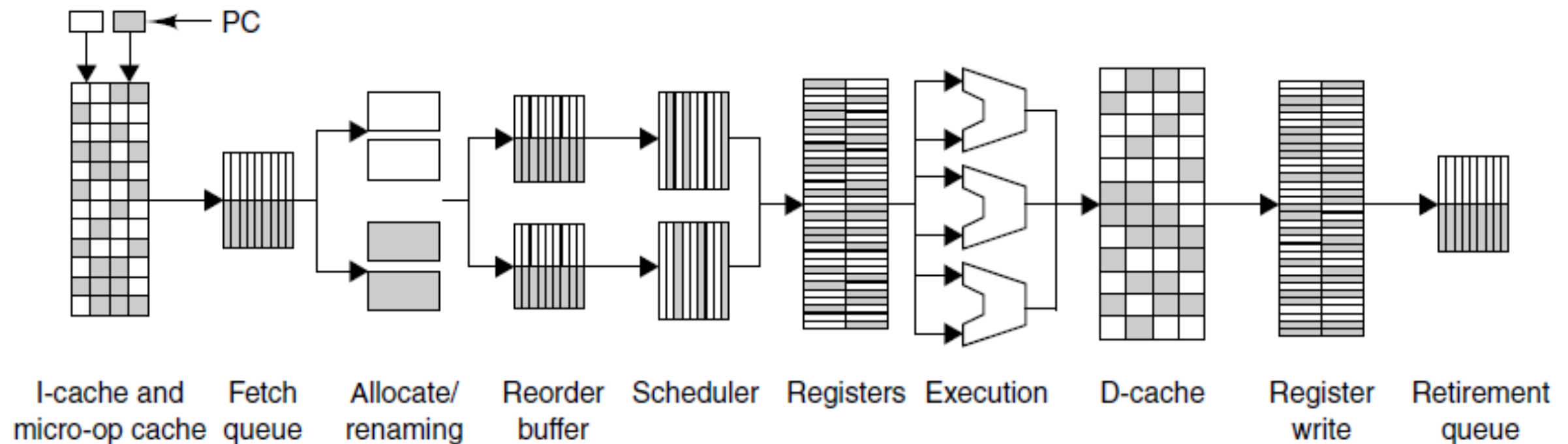
Non-Uniform Memory Access (NUMA):

Time for memory access depends on location of data; also known as Distributed Shared memory machines. Local access is faster than non-local access. Easier to scale than SMPs (e.g.: SGI Origin 2000)





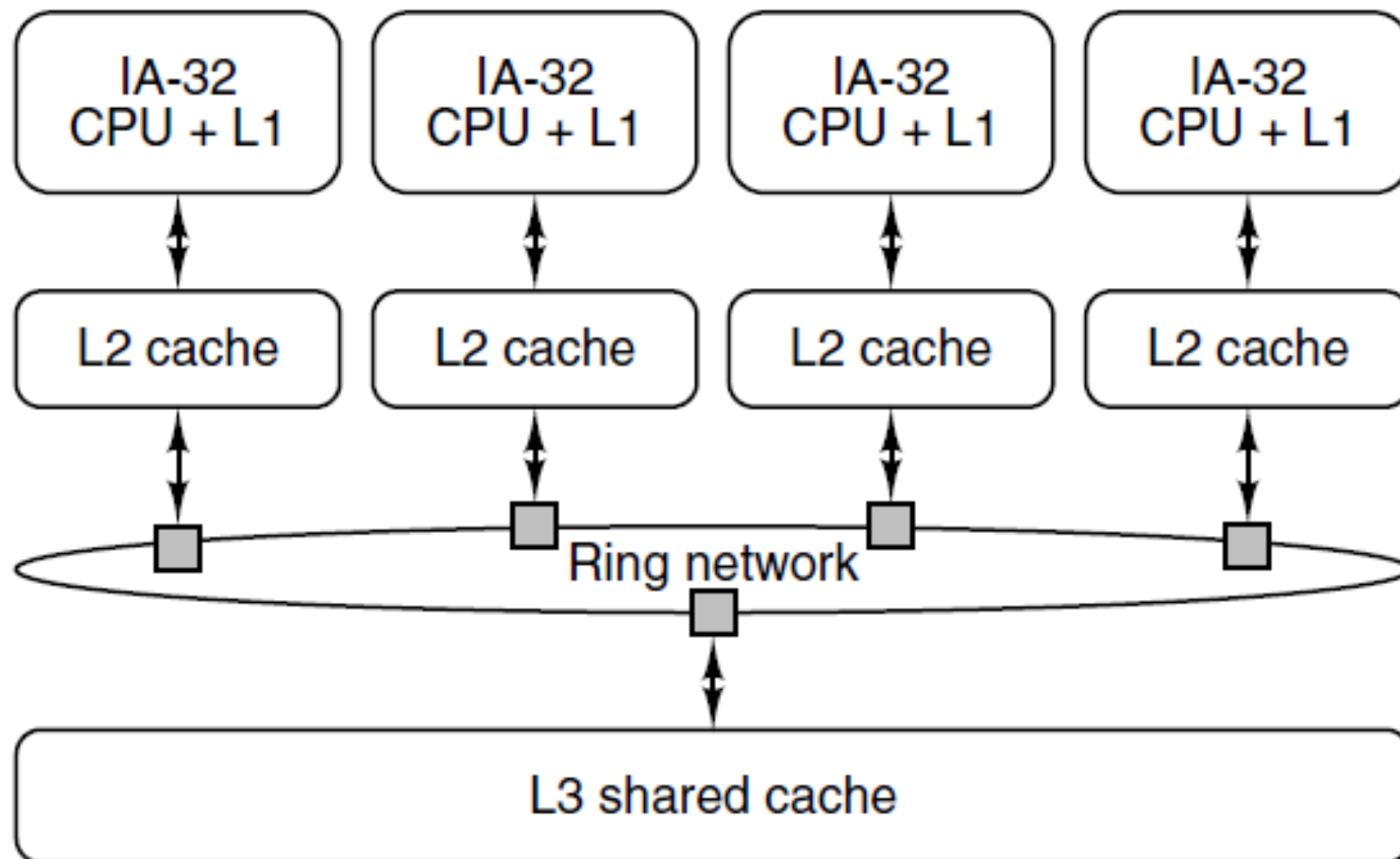
Hyperthreading on the Core i7



Resource sharing between threads in the Core i7 microarchitecture.

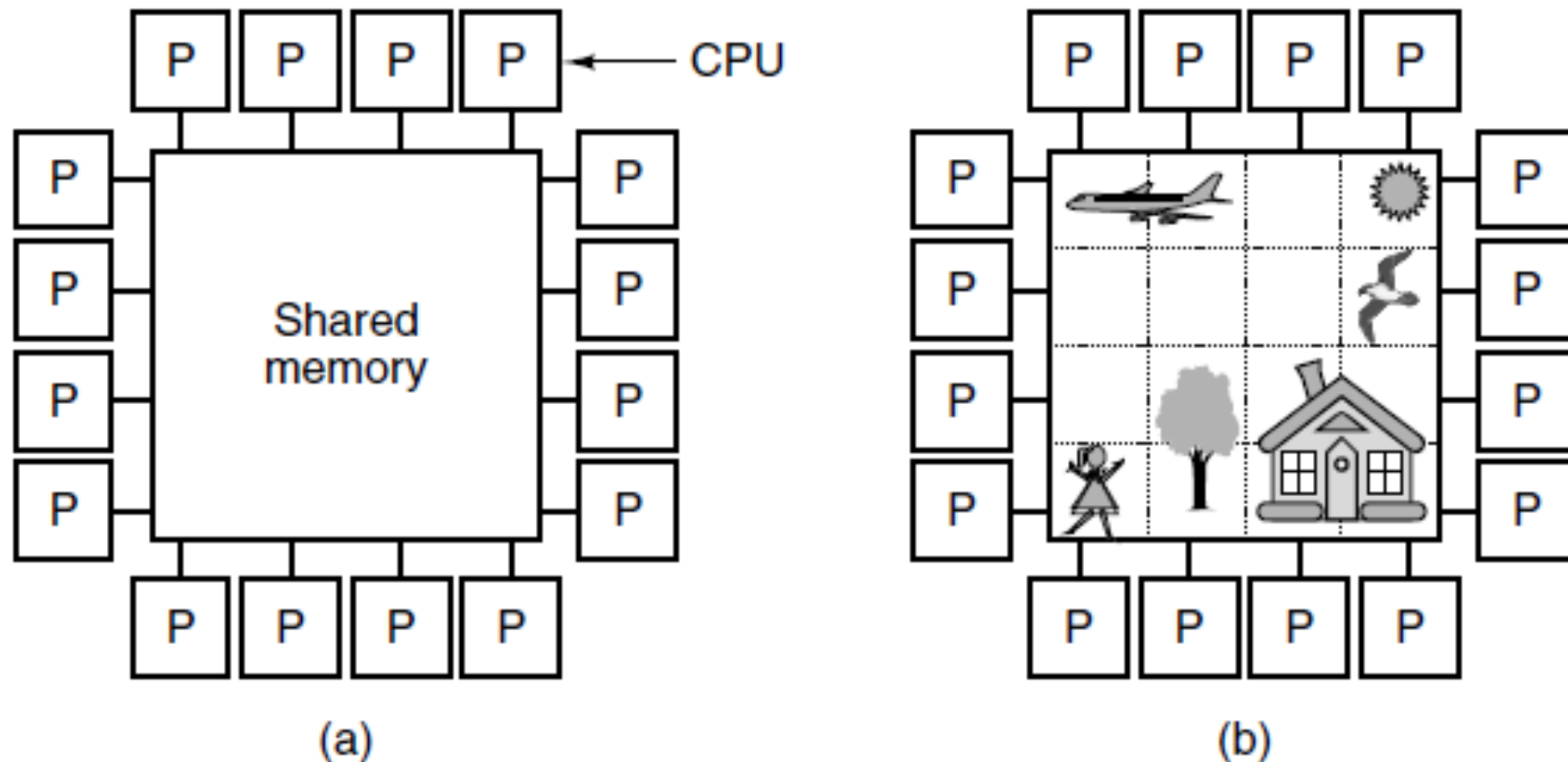


The Core i7 Single-Chip Multiprocessor

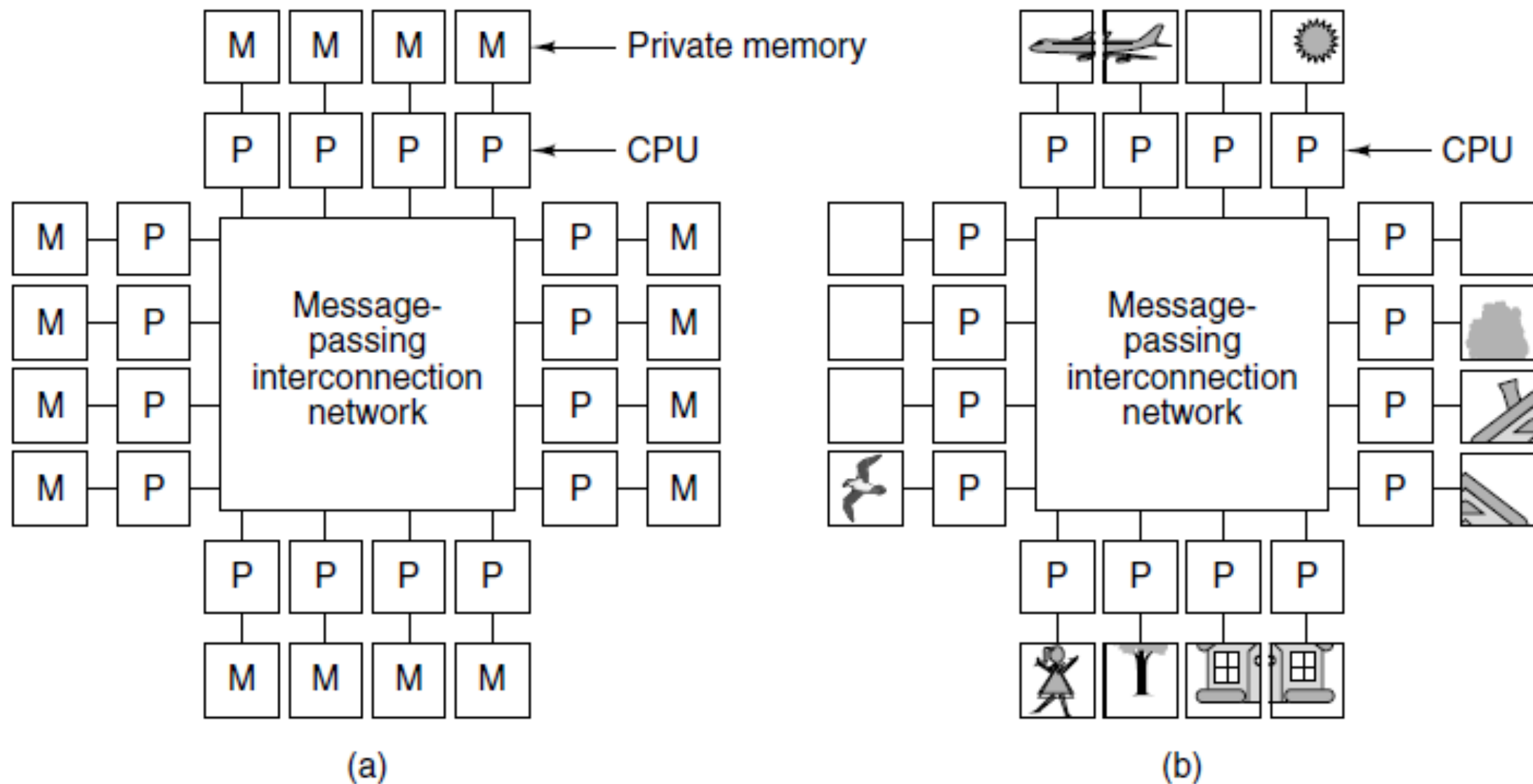


Single-chip multiprocessor architecture of the Core i7.

Multiprocessors



- (a) A multiprocessor with 16 CPUs sharing a common memory.
- (b) An image partitioned into 16 sections, each being analyzed by a different CPU.



- (a) A multicomputer with 16 CPUs, each with its own private memory.
- (b) The bit-map image of split up among the 16 memories.



Principles of Parallel Algorithm Design

Algorithm development is a critical component for solving problems with computers

Sequential algorithm is a sequence of steps to solve the problem

Parallel algorithm is steps to solve problem on multiple processors and threads



Parallel Algorithms

Parallel algorithms involve more than just specifying steps

You have the added dimension of concurrency to deal with. Which bits can execute simultaneously.

This may include any or all of the following:

- Identifying portions of the work that can be performed concurrently

- Mapping the concurrent pieces of work onto multiple processes running in parallel

- Distributing the input, output, and intermediate data associated with the program

- Managing access to data shared by multiple processors

- Synchronising the processors at various stages of the parallel



One size doesn't fit all

Sorry for the bad news folks

HPC programming is hard work with almost no quick fixes. You have to have a good understanding of the underlying computer and the problem you are trying to solve.

Different choices yield best performance on different parallel architectures and different parallel programming paradigms.



Preliminaries

Dividing a computation into smaller components and assigning them to different processors/threads for parallel execution are the two key steps in the design of parallel algorithms.

The process of dividing a computation into smaller parts, some or all of which may be executed in parallel is called *decomposition*.

Tasks are programmer defined units of computation that the main program is divided into by decomposition.

Tasks can be of arbitrary size



Processes vs Processors

Processes are logical computing agents that perform tasks

Processors are the hardware units that physically perform computations.

In Shared Memory Machines we tend to only think in terms of Processes

In Distributed Memory Machines we then to think of processes running of processors



Decomposition Techniques

Recursive Decomposition

The problem is solved by first dividing it into a set of independent subproblems.

Each of these subproblems is solved by recursively applying a similar division into smaller sub-problems

Data Decomposition

Step 1 the data on which the computations are performed is partitioned

Step 2 this data partitioning is used to induce a partitioning of computations into tasks

Exploratory Decomposition

We partition the search space into smaller parts and search each one of these parts concurrently



Characteristics of Sequential and Multithreaded Programs

Sequential Programs	Multithreaded Programs
Can only utilise a single core	Can take advantage of multicore multiprocessor hardware
Behaviour is usually deterministic	Behaviour is usually nondeterministic
Testing that covers all code finds most bugs	Code coverage alone is insufficient; threads racing to access memory cause the most problematic bugs
Root causes for failures are found by carefully tracing execution that leads to failure	Root causes are found by postulating a race that fits the behaviour and methodically reviewing the code for possible causes
Once identified, finding a fix for a bug is generally assured	The root cause of a race can easily remain unidentified



Programming in Sequential and

Sequential Programs	Multithreaded Programs
Memory is stable unless explicitly updated	Memory is in flux unless it is read-only, thread local, or protected by a lock
No locks are needed	Locks are essential
No special considerations when accessing two interdependent data structures	If two data structures are related, locks for both structures must be entered before using that relationship
Deadlock can't happen	Deadlock is possible when there are multiple, unordered locks

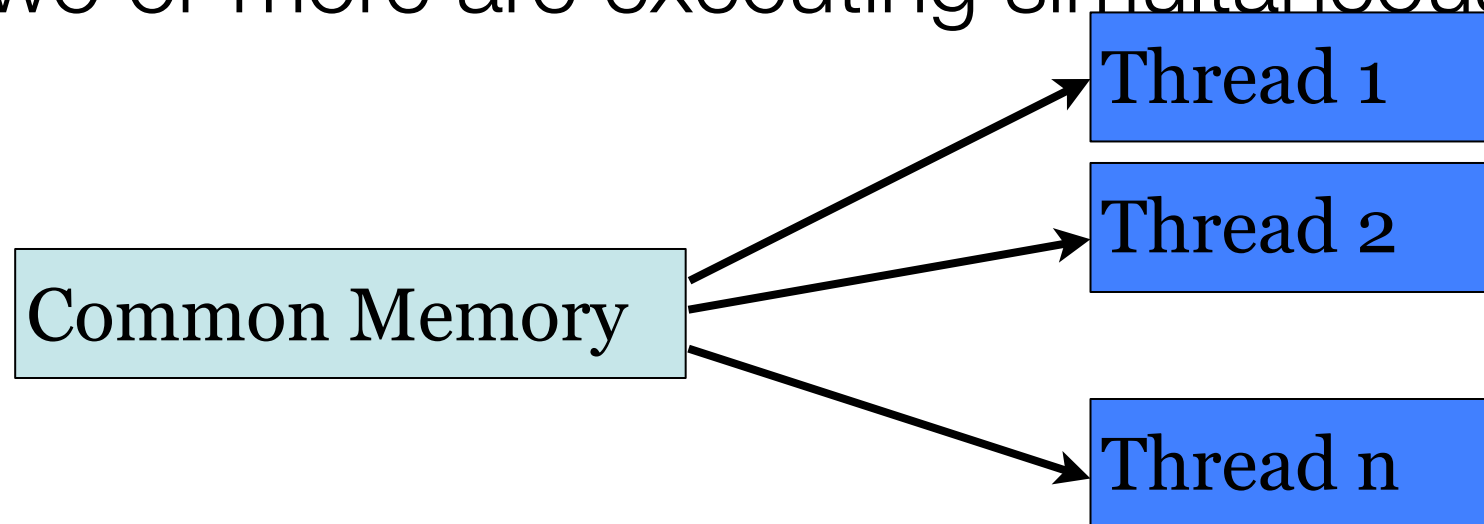


Threads and Memory

Seems simple enough

Instead of one processing unit doing work sequentially

Two or more are executing simultaneously



Advantage is multithreaded programs are programmed in a similar way to sequential programs

BUT the model does not distinguish between local and shared memory



Four conditions needed for a race to be possible

1. There are memory locations that are accessible from more than one thread
2. There is a property associated with these shared memory locations that is needed for the program to function properly
3. The property does not hold during some part of the update.
4. Another thread must access the memory when the invariant is broken



Races are a pain in the ...

Races are the biggest enemy of shared memory code

The biggest slow down of code is protecting against them



Locks

The most common way of preventing races is to use locks to prevent other threads from accessing memory associated with an invariant while it is broken.

The same basic functionality – lots of different names

- a monitor

- a critical section

- a mutex

- a semaphore

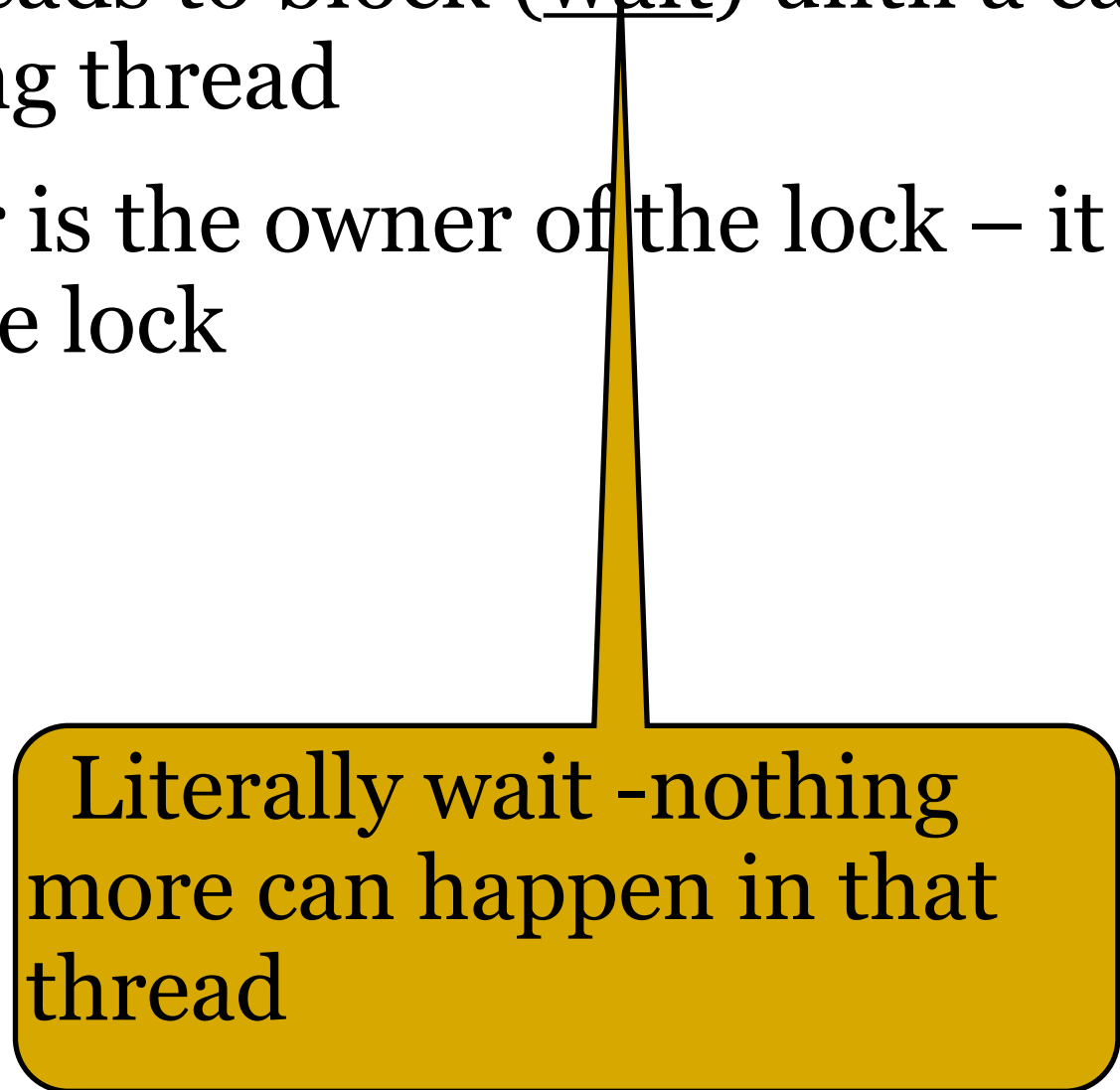


How Locks Work

The lock provides an Enter and Exit method

Once a thread calls Enter, all attempts by other threads to call Enter will cause the threads to block (wait) until a call to Exit is made by the locking thread

The thread that called Enter is the owner of the lock – it MUST call Exit to release the lock



Literally wait -nothing more can happen in that thread



How many locks

`totalRequests = highPriRequests + lowPriRequests`

One or two locks?

Fewer locks – simpler & perhaps slower

More locks – more complexity, but generally faster



Deadlock

Once a program has more than one lock, deadlock becomes a possibility

Thread 1 tries to Enter lock A

Thread 2 tries to Enter lock B

Thread 1 tries to Enter lock B – WAIT

Thread 2 tries to Enter lock A – WAIT

Twiddle thumbs until the end of time or the process is killed



Preventing Deadlock

Best method don't have too many locks – so it is never necessary to take out more than one lock at a time

If you can't prevent that – have a convention on the order in which locks are taken.



Cost of Locks

The lightest locks use a special compare/exchange instruction to check if the lock is taken

If it isn't they enter the lock in a single atomic instruction

This special instruction is expensive (10 – 100x) more than an ordinary instruction



Cost of Locks

Two main reasons for this.

1. The compare/exchange instruction must ensure that no other thread or processor is also trying to do the same thing.
This requires one processor to co-ordinate with all other processors in the system.
2. The effect inter-process communication has. After a lock has been taken, the program is very likely to access memory that may have recently been modified by another thread. If this thread was on another processor, it is necessary to ensure all pending writes are flushed.



It's a bit like Goldilocks & the 3 Bears

Just like Goldilocks you'll have to try various options. Some solutions will be:

Too cold – not enough locks the system is waiting too much to gain access to critical sections

Too hot – too many locks the system is spending too much time acquiring and releasing locks

Just right – what we dream of





Optimising Code

Don't do it until you need to

Compilers are very good at doing this, but YOU have more understanding of how things work.

If you can use tools to help you - Intel suite is good



References

Many slides taken from the following resource packs

- Tanenbaum, Structured Computer Organization, Fifth Edition, (c) 2006 Pearson Education, Inc. All rights reserved
- Victor Eijkhout, Texas Advanced Computing Center, The University of Texas.

These are great resources and I thank the authors for making them publicly available and giving me permission to use them.