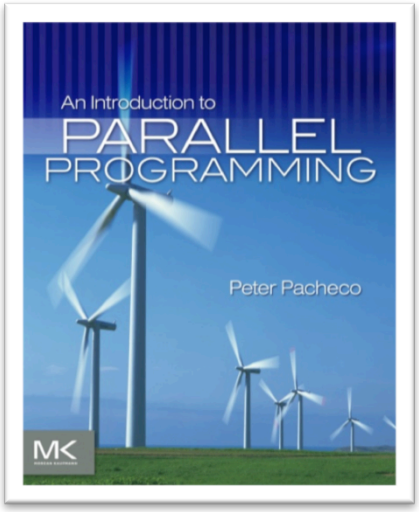
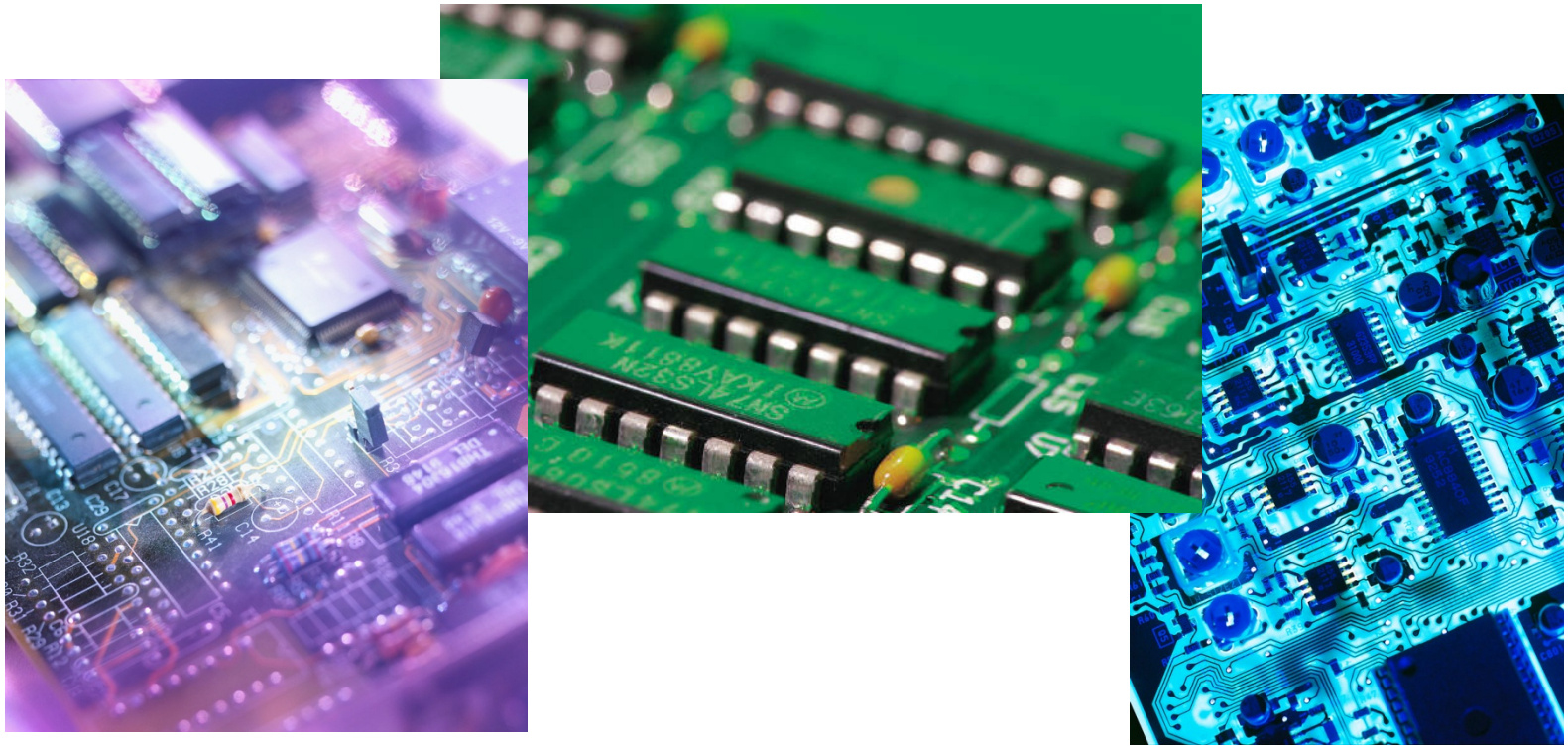


Chapter 2

Parallel Hardware and Parallel Software

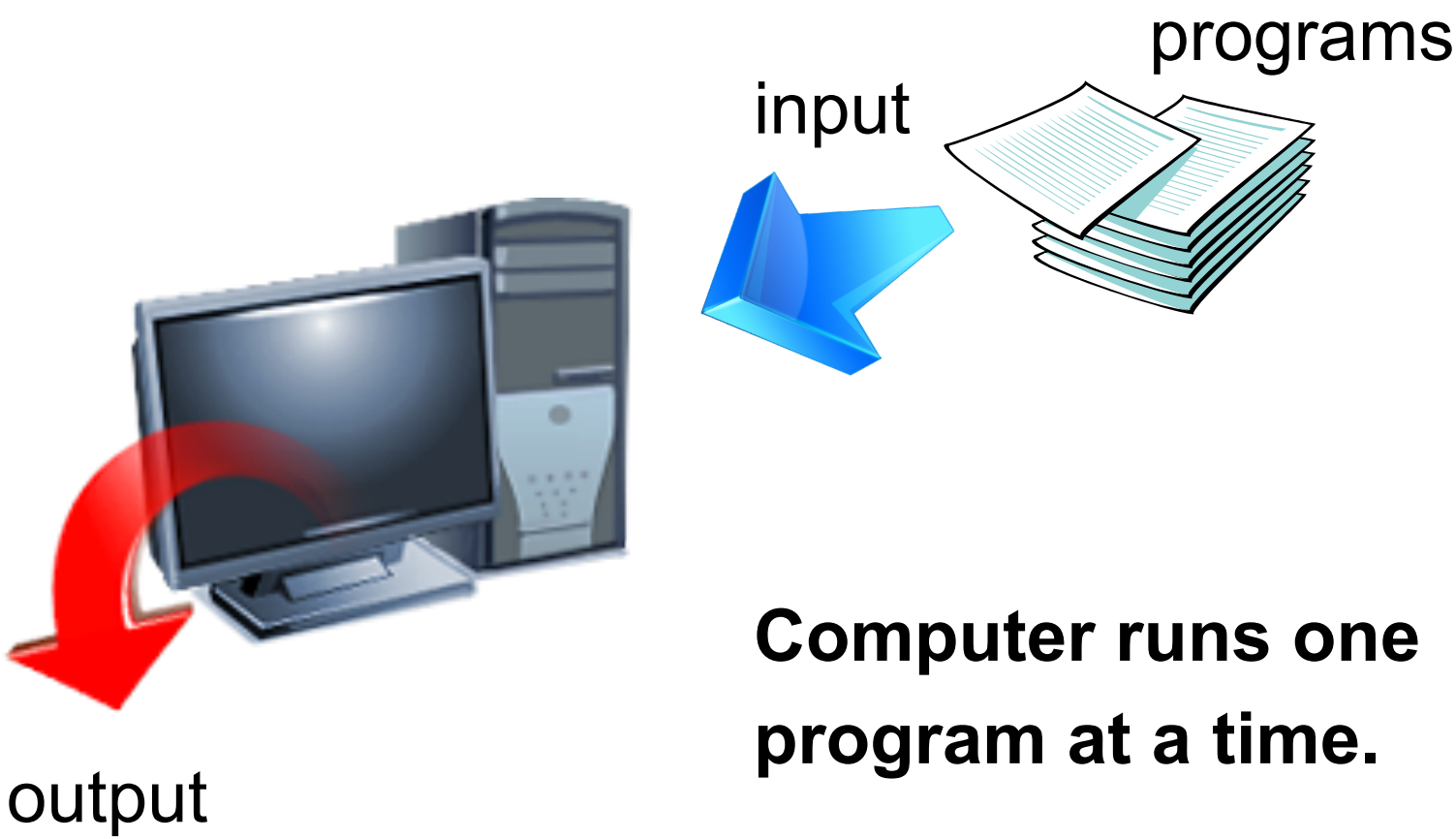
Roadmap

- Some background
- Modifications to the von Neumann model
- Parallel hardware
- Performance
  - Using Linux tools
- Starting with Pthreads (chapter 4)



SOME BACKGROUND

Serial hardware and software



Computer runs one program at a time.

The von Neumann Architecture

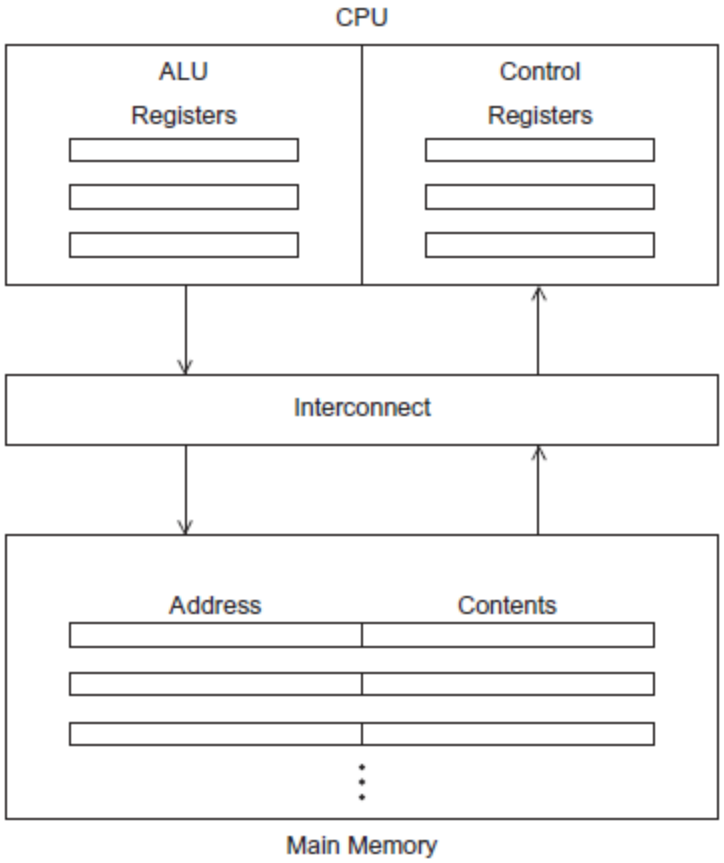
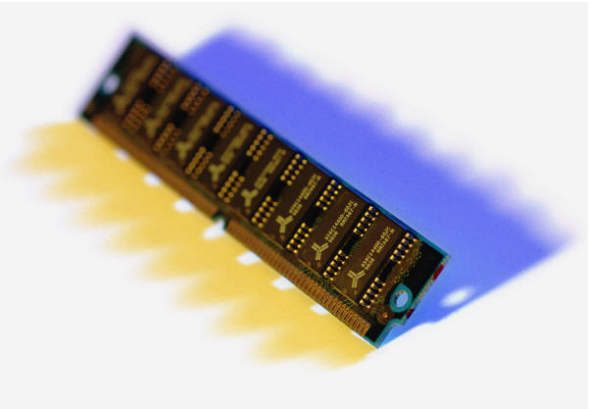


Figure 2.1

Main memory

- This is a collection of locations, each of which is capable of storing both instructions and data.
- Every location consists of an address, which is used to access the location, and the contents of the location.



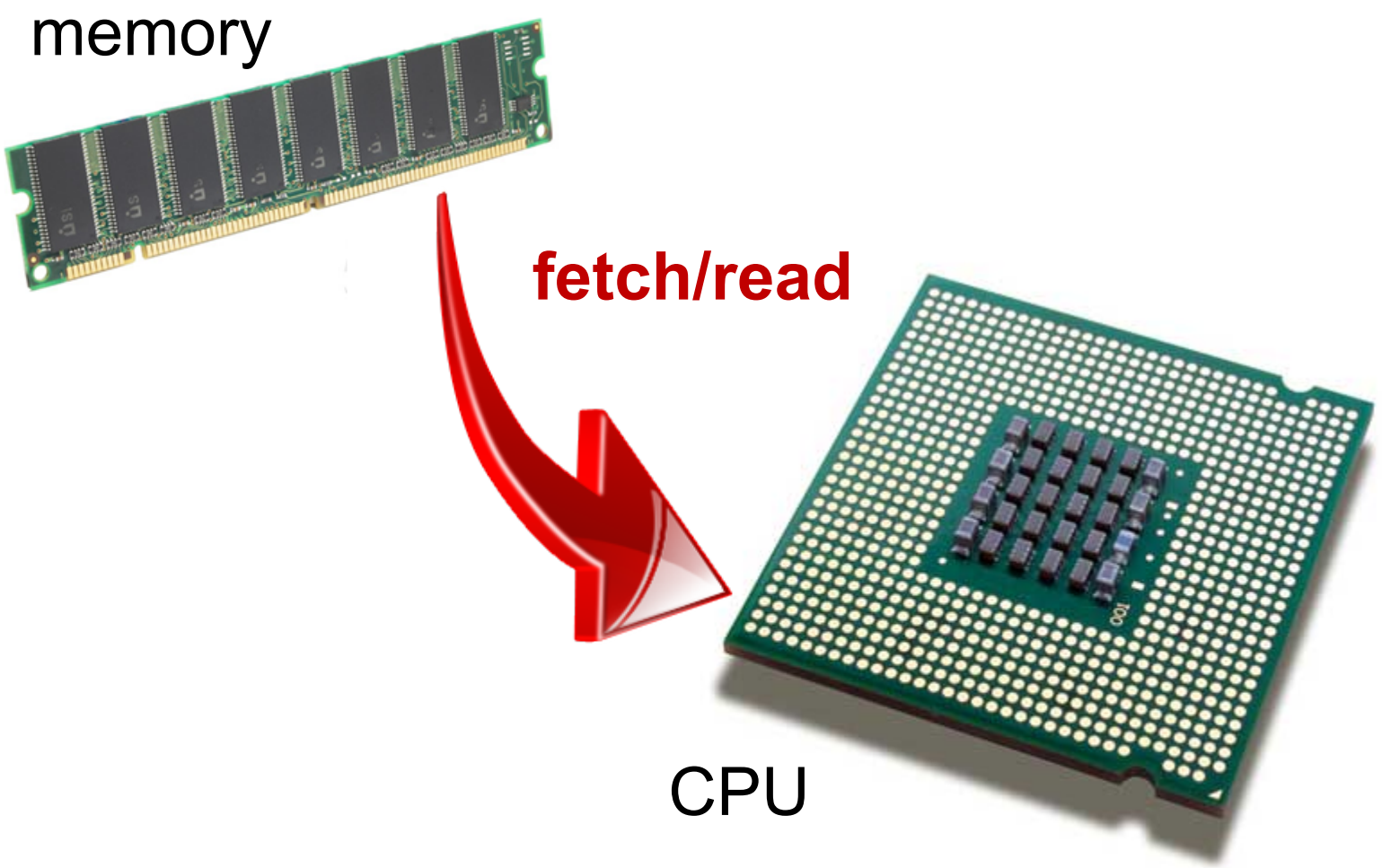
Central processing unit (CPU)

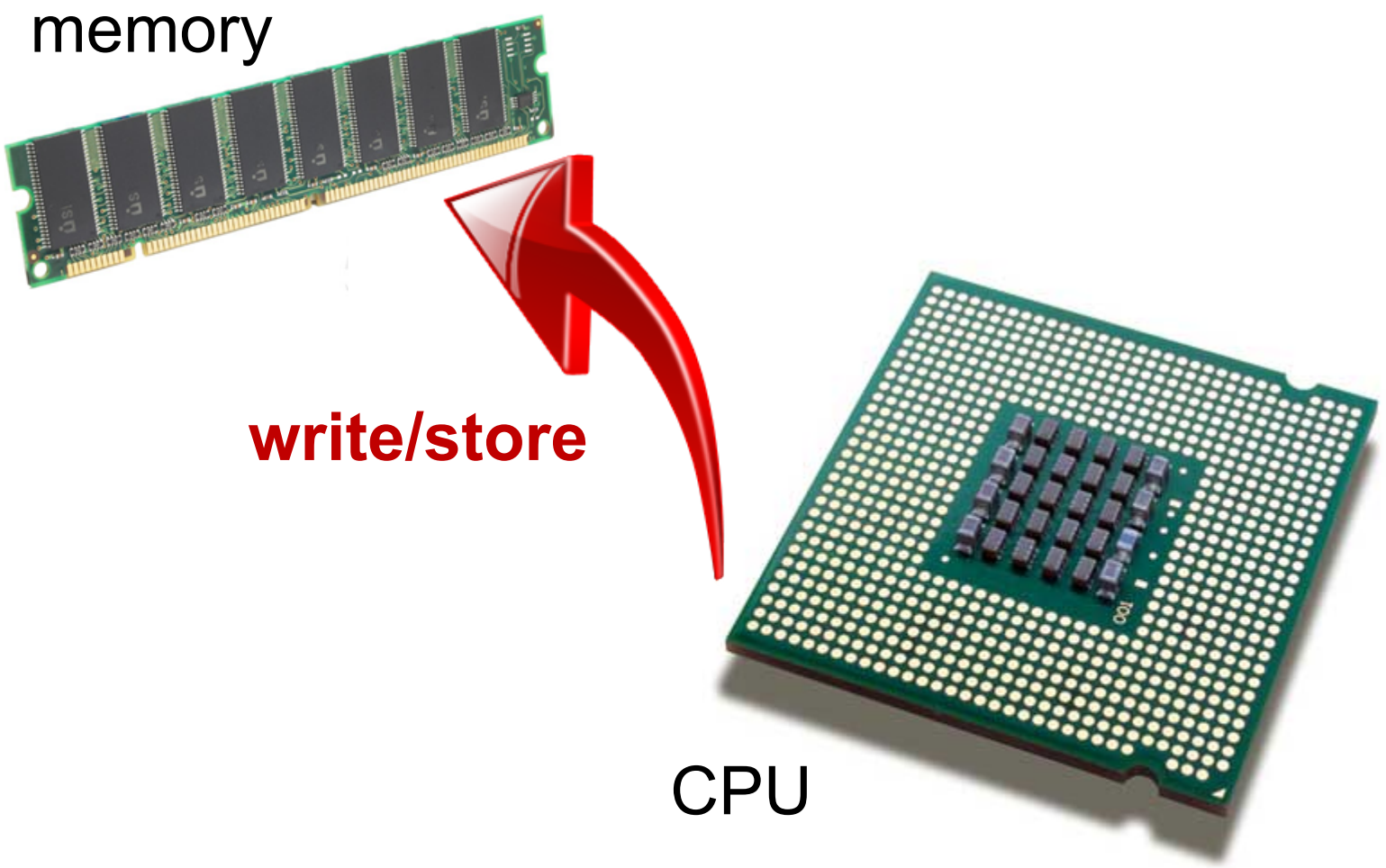
- Divided into two parts.
- **Control unit** - responsible for deciding which instruction in a program should be executed. (*the boss*)

- **Arithmetic and logic unit** (ALU) - responsible for executing the actual instructions. (*the worker*)

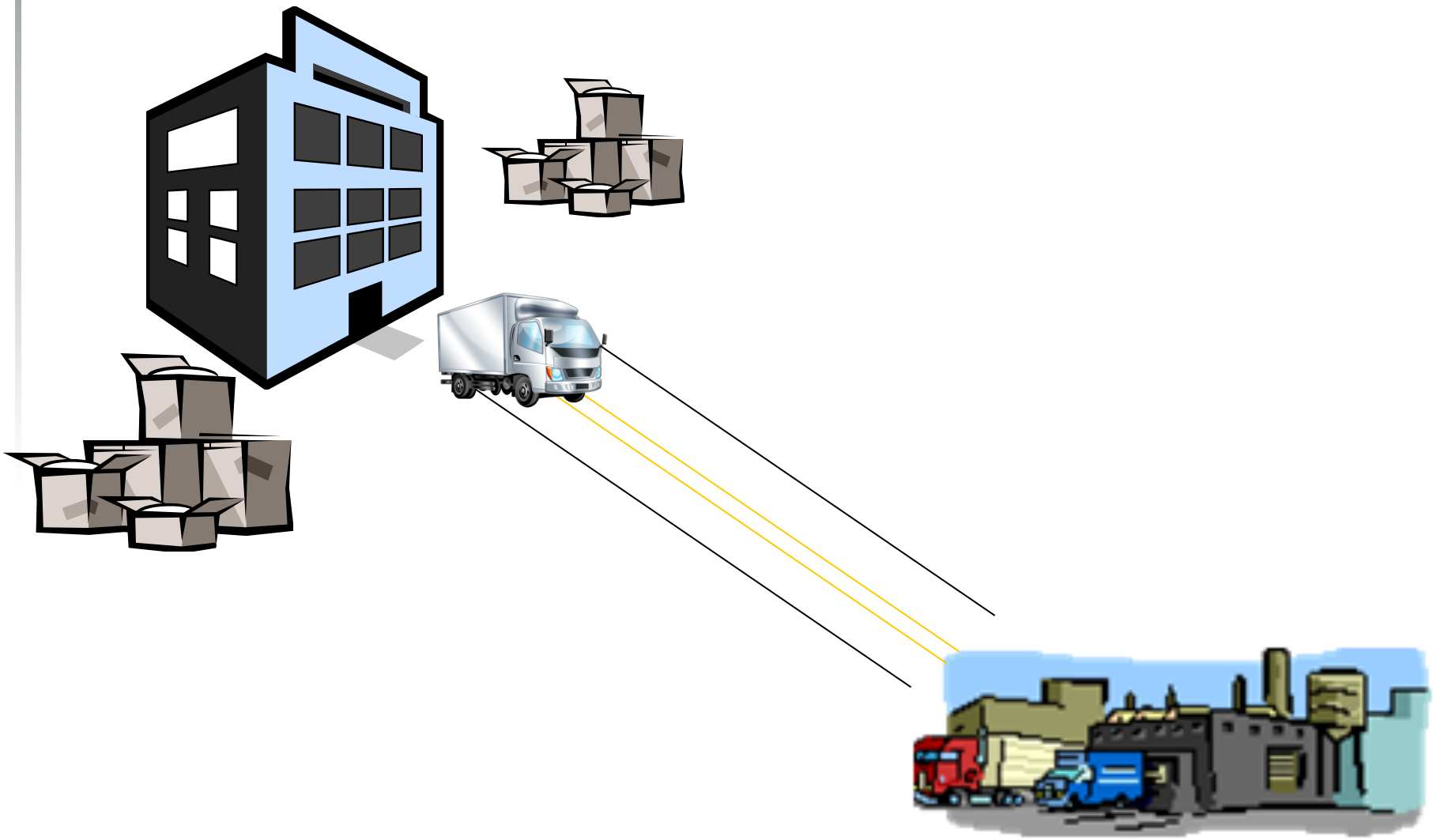
Key terms

- **Register** – very fast storage, part of the CPU.
- **Program counter** – stores address of the next instruction to be executed.
- **Bus** – wires and hardware that connects the CPU and memory.





von Neumann bottleneck



An operating system "process"

- An instance of a computer program that is being executed.
- Components of a process:
  - The executable machine language program.
  - A block of memory.
  - Descriptors of resources the OS has allocated to the process.
  - Security information.
  - Information about the state of the process.

Multitasking

- Gives the illusion that a single processor system is running multiple programs simultaneously.
- Each process takes turns running. (**time slice**)
- After its time is up, it waits until it has a turn again. (**blocks**)

Threading

- Threads are contained within processes.
- They allow programmers to divide their programs into (more or less) independent tasks.
- The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run.

A process and two threads

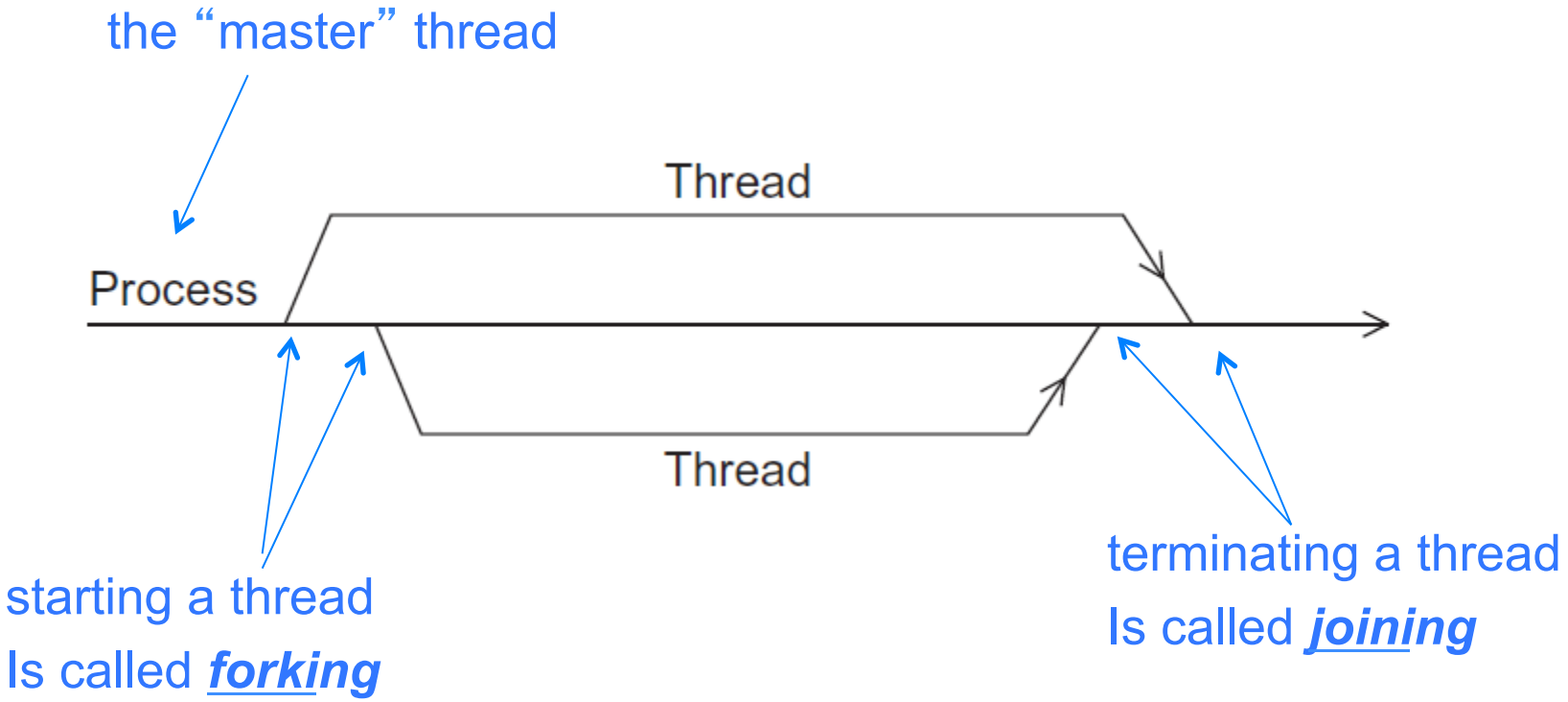
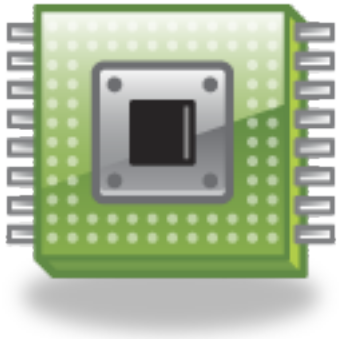


Figure 2.2



MODIFICATIONS TO THE VON NEUMANN MODEL

Basics of caching

- A collection of memory locations that can be accessed in less time than some other memory locations.
- A CPU cache is typically located on the same chip, or one that can be accessed much faster than ordinary memory.

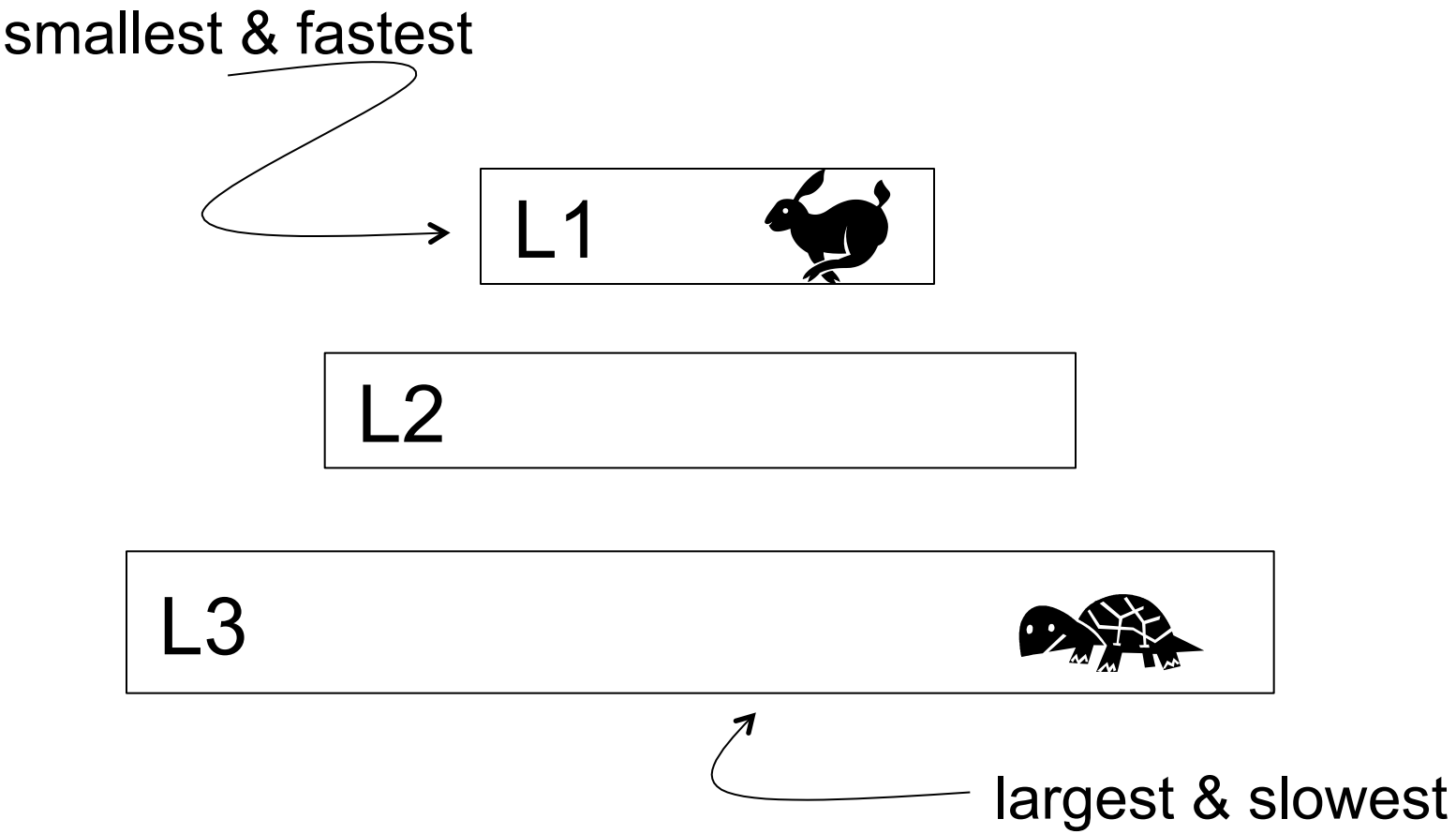
Principle of locality

- Accessing one location is followed by an access of a nearby location.
- **Spatial locality** – accessing a nearby location.
- **Temporal locality** – accessing in the near future.

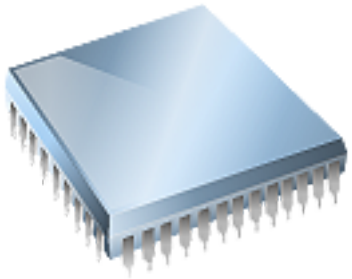
Principle of locality

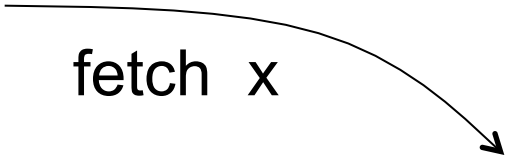
```
float z[1000];
...
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

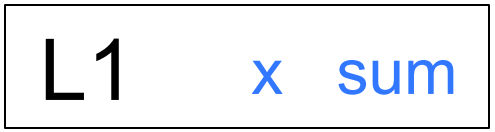
Levels of Cache

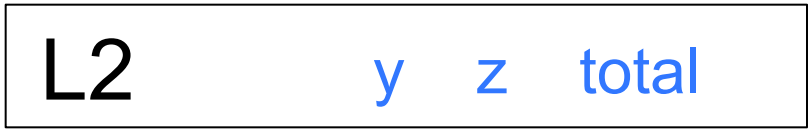


Cache hit

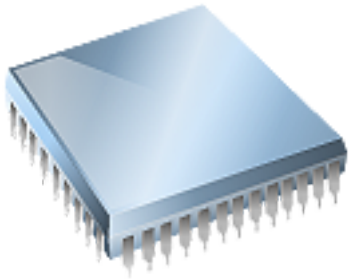




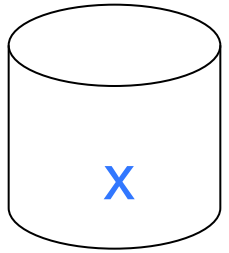




Cache miss



fetch x



main memory

L1 y sum

L2 r1 z total

L3 A[ ] radius center

Issues with cache

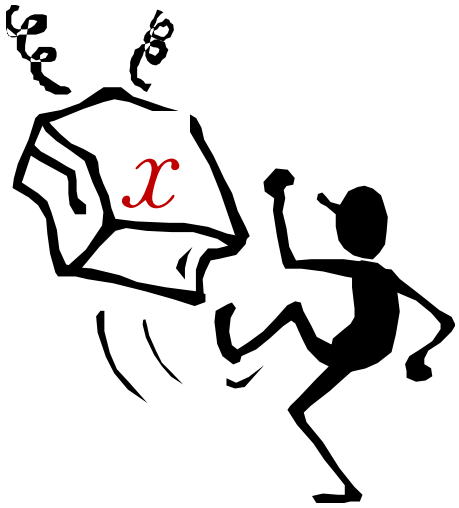
- When a CPU writes data to cache, the value in cache may be inconsistent with the value in main memory.
- **Write-through** caches handle this by updating the data in main memory at the time it is written to cache.
- **Write-back** caches mark data in the cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.

Cache mappings

- **Full associative** – a new line can be placed at any location in the cache.
- **Direct mapped** – each cache line has a unique location in the cache to which it will be assigned.
- ***n*-way set associative** – each cache line can be place in one of *n* different locations in the cache.

n-way set associative

- When more than one line in memory can be mapped to several different locations in cache we also need to be able to decide which line should be replaced or evicted.



Example

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Table 2.1: Assignments of a 16-line main memory to a 4-line cache

Caches and programs

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
        y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
    for (i = 0; i < MAX; i++)
        y[i] += A[i][j]*x[j];
```

Caches and programs

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
        y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
    for (i = 0; i < MAX; i++)
        y[i] += A[i][j]*x[j];
```

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

Virtual memory (1)

- If we run a very large program or a program that accesses very large data sets, all of the instructions and data may not fit into main memory.
- Virtual memory functions as a cache for secondary storage.

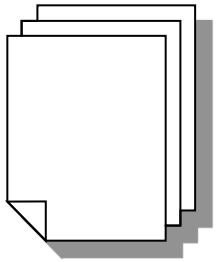
Virtual memory (2)

- It exploits the principle of spatial and temporal locality.
- It only keeps the active parts of running programs in main memory.

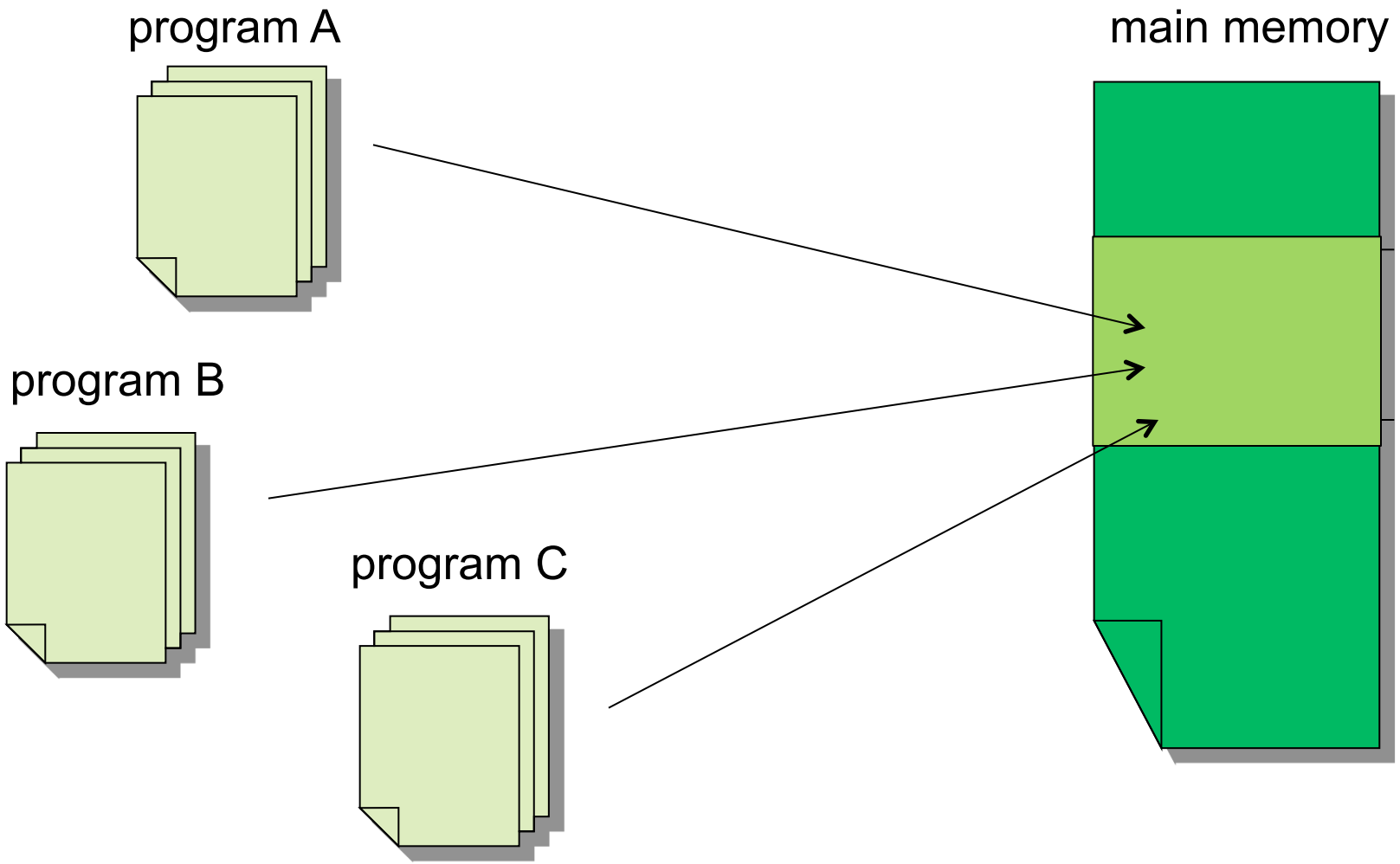
Virtual memory (3)

- **Swap space** - those parts that are idle are kept in a block of secondary storage.

- **Pages** – blocks of data and instructions.
  - Usually these are relatively large.
  - Most systems have a fixed page size that currently ranges from 4 to 16 kilobytes.



Virtual memory (4)



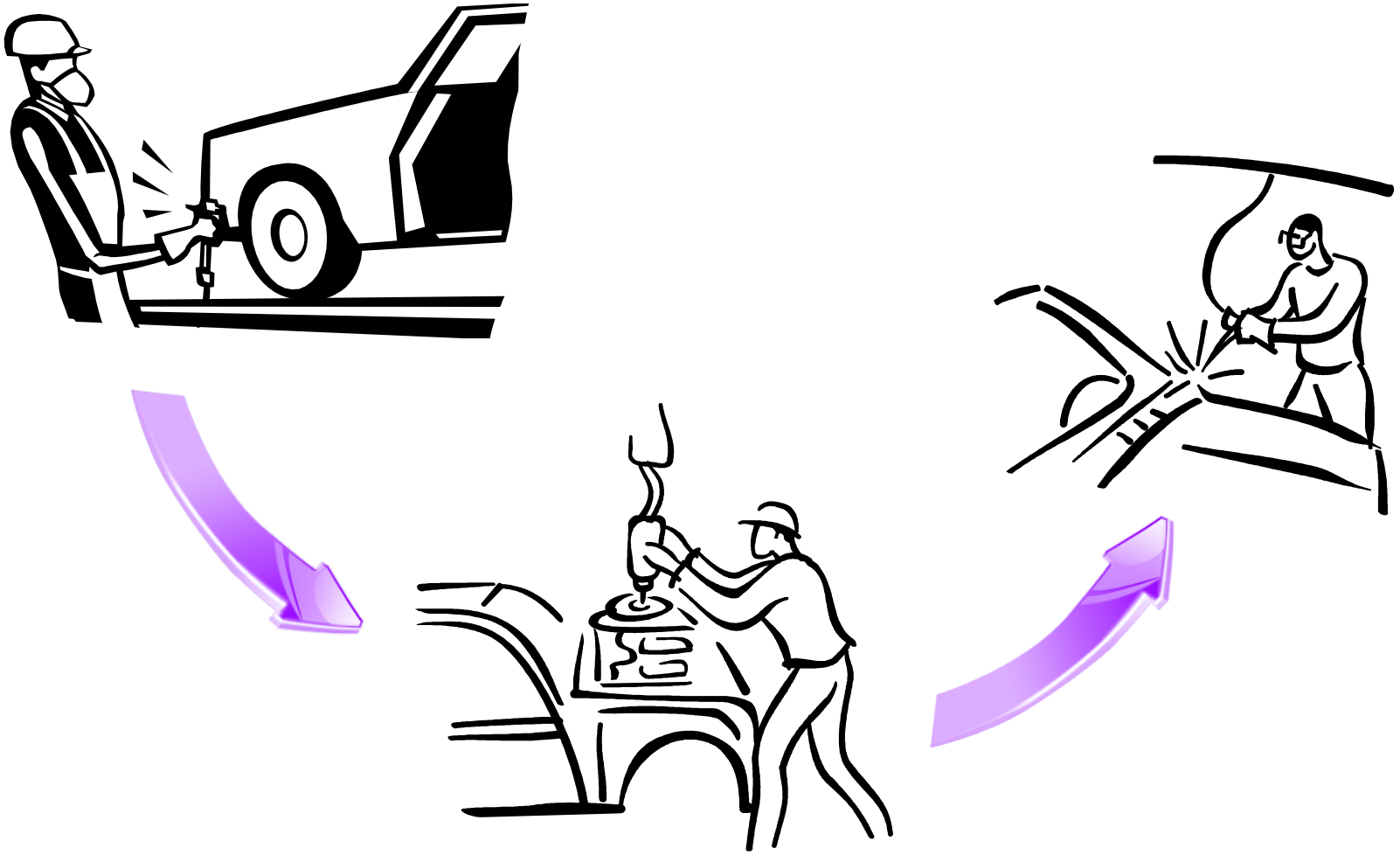
Instruction Level Parallelism (ILP)

- Attempts to improve processor performance by having multiple processor components or **functional units** simultaneously executing instructions.

Instruction Level Parallelism (2)

- **Pipelining** - functional units are arranged in stages.
- **Multiple issue** - multiple instructions can be simultaneously initiated.

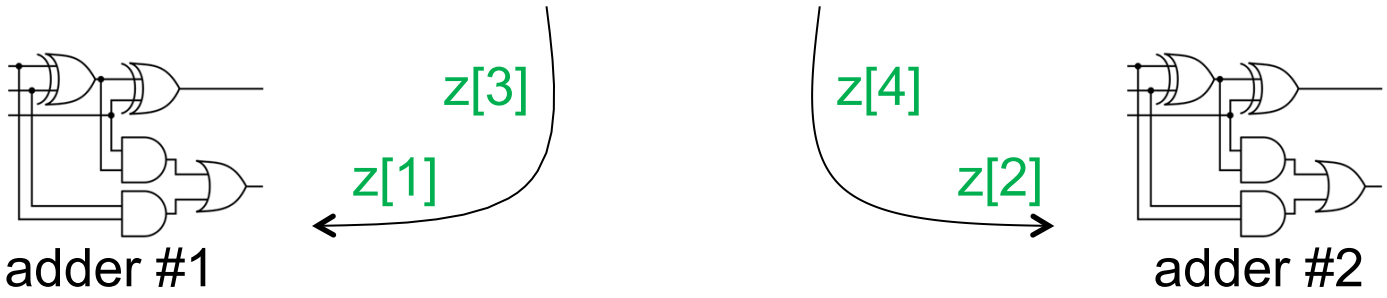
Pipelining



Multiple Issue (1)

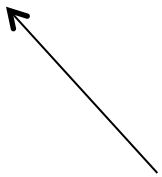
- Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program.

```
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```

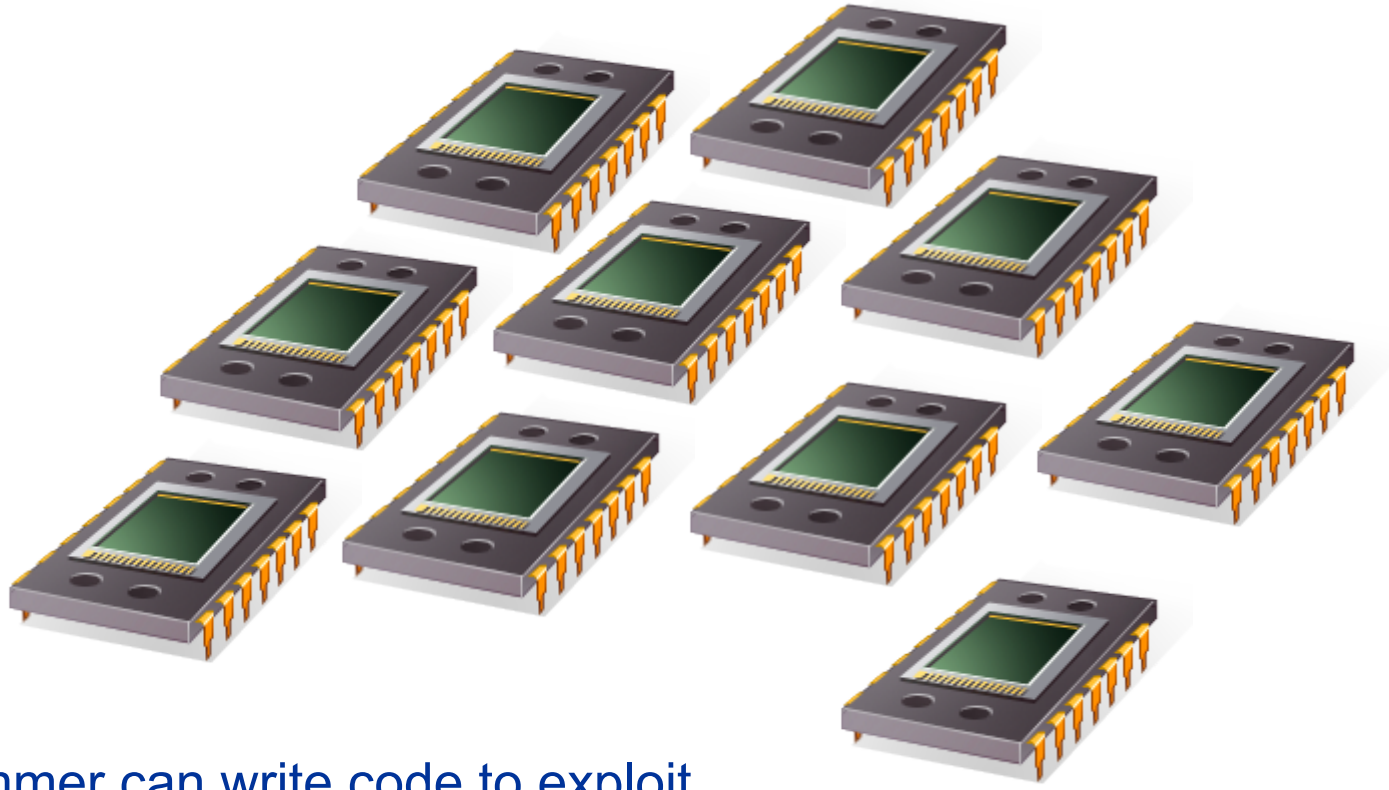


Multiple Issue (2)

- **static** multiple issue - functional units are scheduled at compile time.
- **dynamic** multiple issue – functional units are scheduled at run-time.



superscalar



A programmer can write code to exploit.

PARALLEL HARDWARE

Flynn's Taxonomy

SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream	(MIMD) Multiple instruction stream Multiple data stream

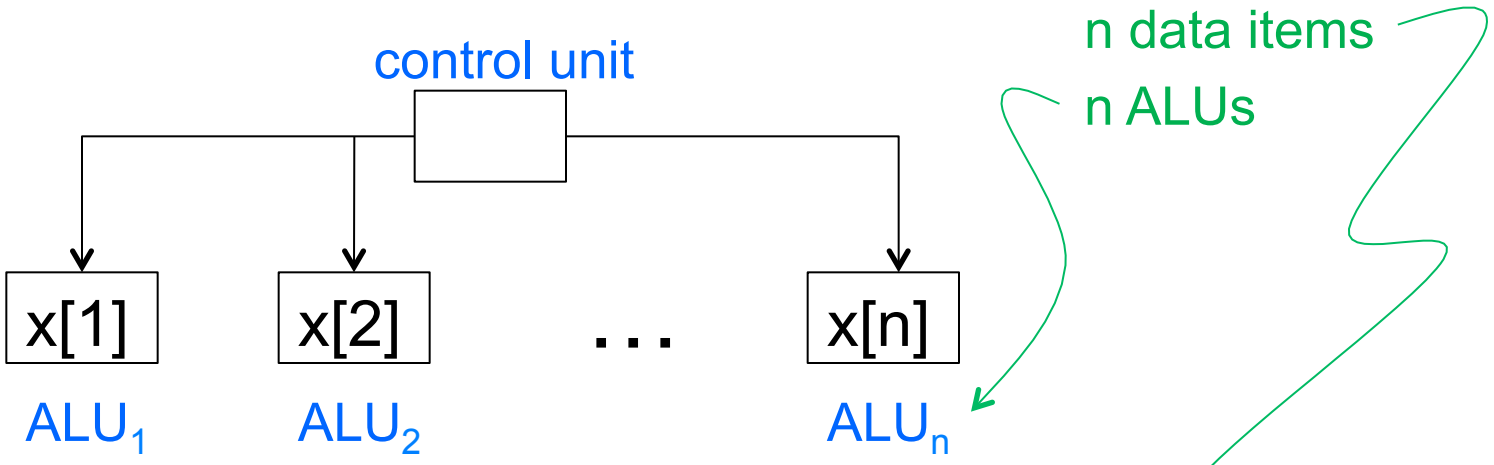
classic von Neumann

not covered

SIMD

- Parallelism achieved by dividing data among the processors.
- Applies the same instruction to multiple data items.
- Called data parallelism.

SIMD example




```
for (i = 0; i < n; i++)
    x[i] += y[i];
```

SIMD

- What if we don't have as many ALUs as data items?
- Divide the work and process iteratively.
- Ex. m = 4 ALUs and n = 15 data items.

Round3	ALU_1	ALU_2	ALU_3	ALU_4
1	X[0]	X[1]	X[2]	X[3]
2	X[4]	X[5]	X[6]	X[7]
3	X[8]	X[9]	X[10]	X[11]
4	X[12]	X[13]	X[14]	

SIMD drawbacks

- All ALUs are required to execute the same instruction, or remain idle.
- In classic design, they must also operate synchronously.
- The ALUs have no instruction storage.
- Efficient for large data parallel problems, but not other types of more complex parallel problems.

MIMD

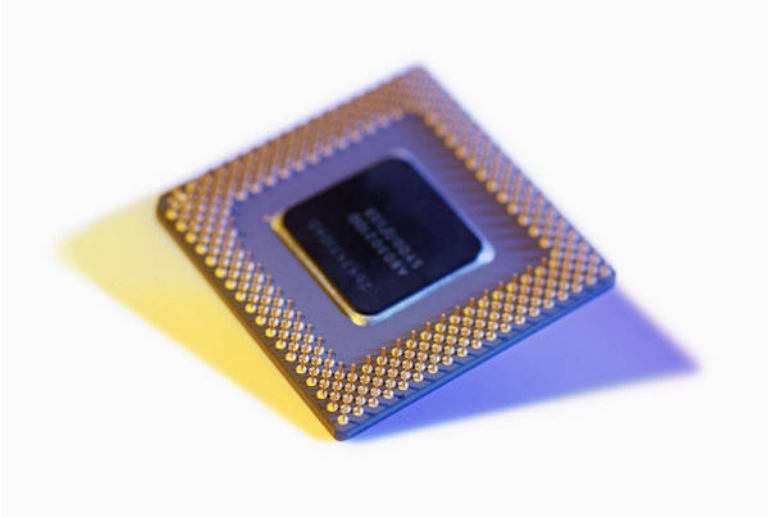
- Supports multiple simultaneous instruction streams operating on multiple data streams.
- Typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.

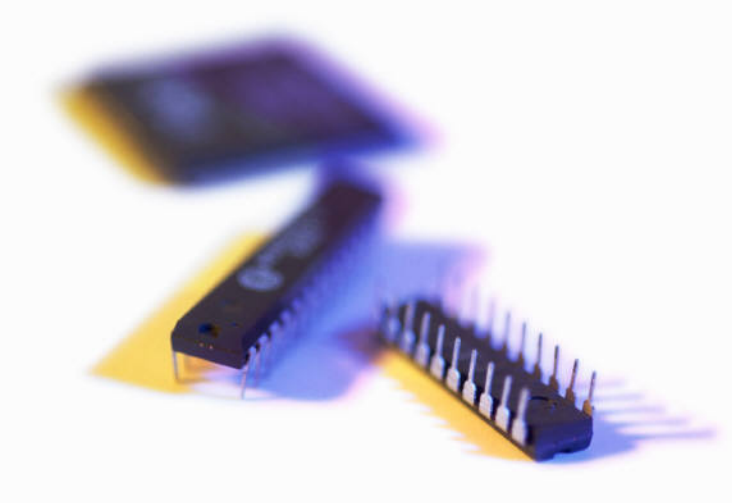
Shared Memory System (1)

- A collection of autonomous processors is connected to a memory system via an interconnection network.
- Each processor can access each memory location.
- The processors usually communicate implicitly by accessing shared data structures.

Shared Memory System (2)

- Most widely available shared memory systems use one or more multicore processors.
  - (multiple CPU's or cores on a single chip)





Shared Memory System

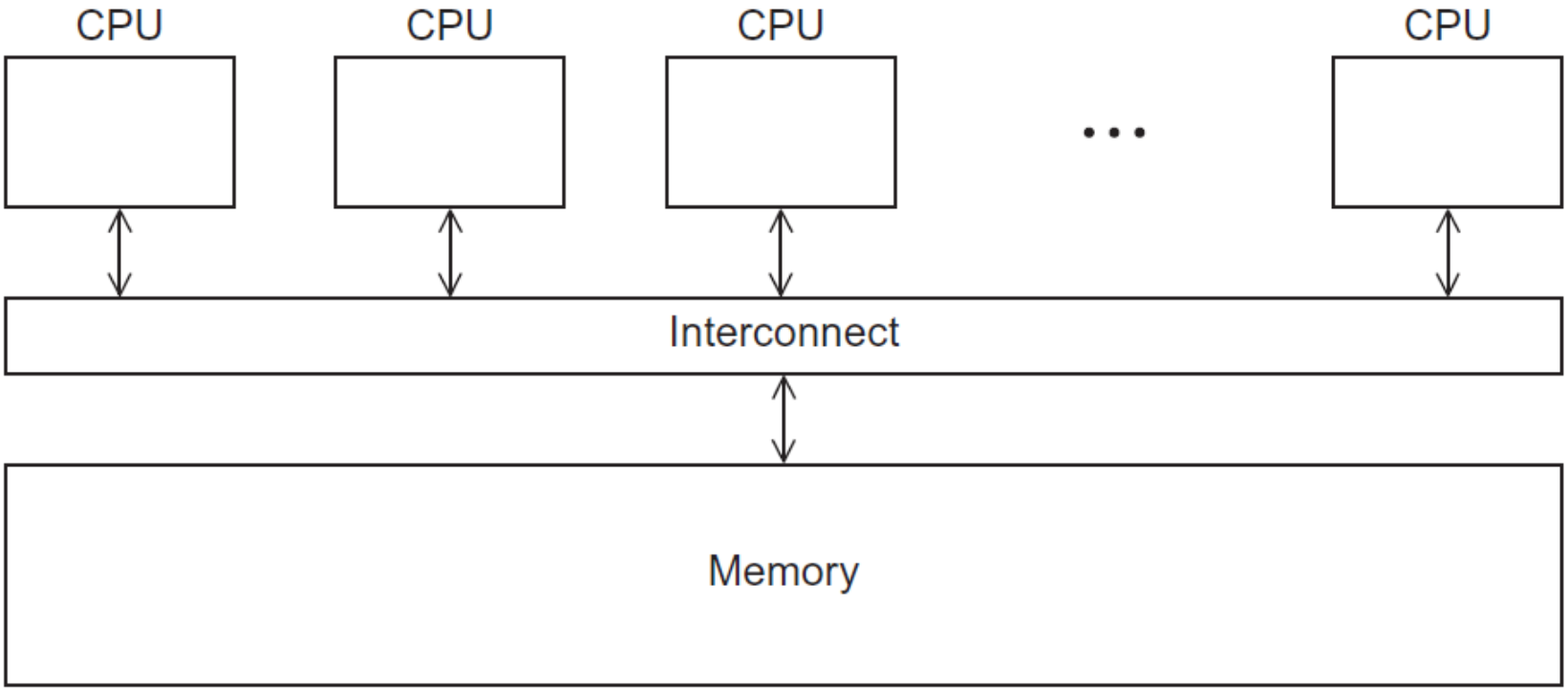
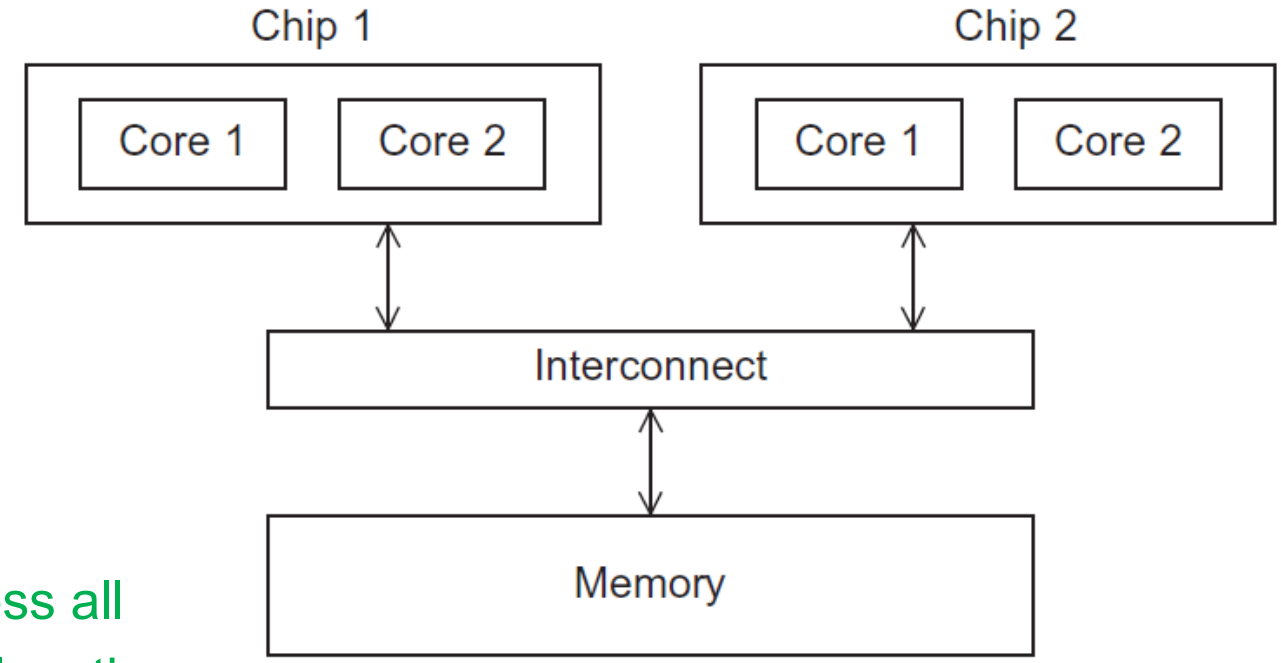


Figure 2.3

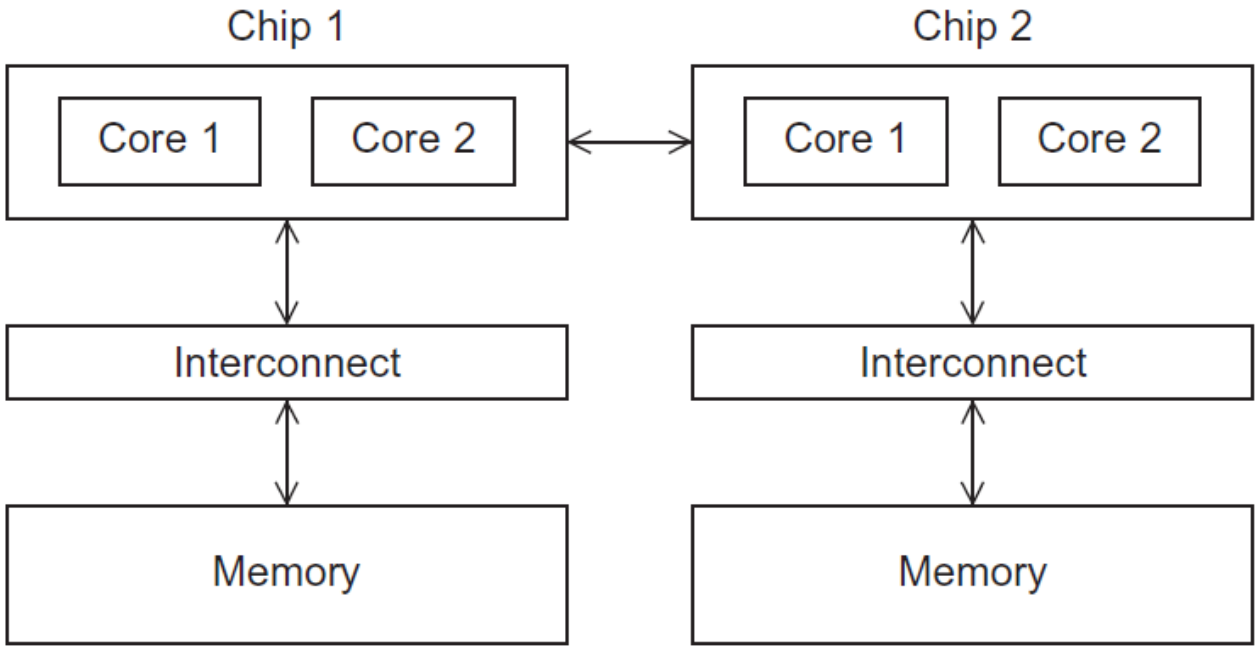
UMA multicore system



Time to access all the memory locations will be the same for all the cores.

Figure 2.5

NUMA multicore system



A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

Figure 2.6

PERFORMANCE

Speedup

- Number of cores = p
- Serial run-time = T_{serial}
- Parallel run-time = $T_{parallel}$

linear speedup

$$T_{parallel} = T_{serial} / p$$

Speedup of a parallel program

$$S = \frac{T_{serial}}{T_{parallel}}$$

Efficiency of a parallel program

$$E = \frac{S}{p} = \frac{\left(\frac{T_{serial}}{T_{parallel}}\right)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

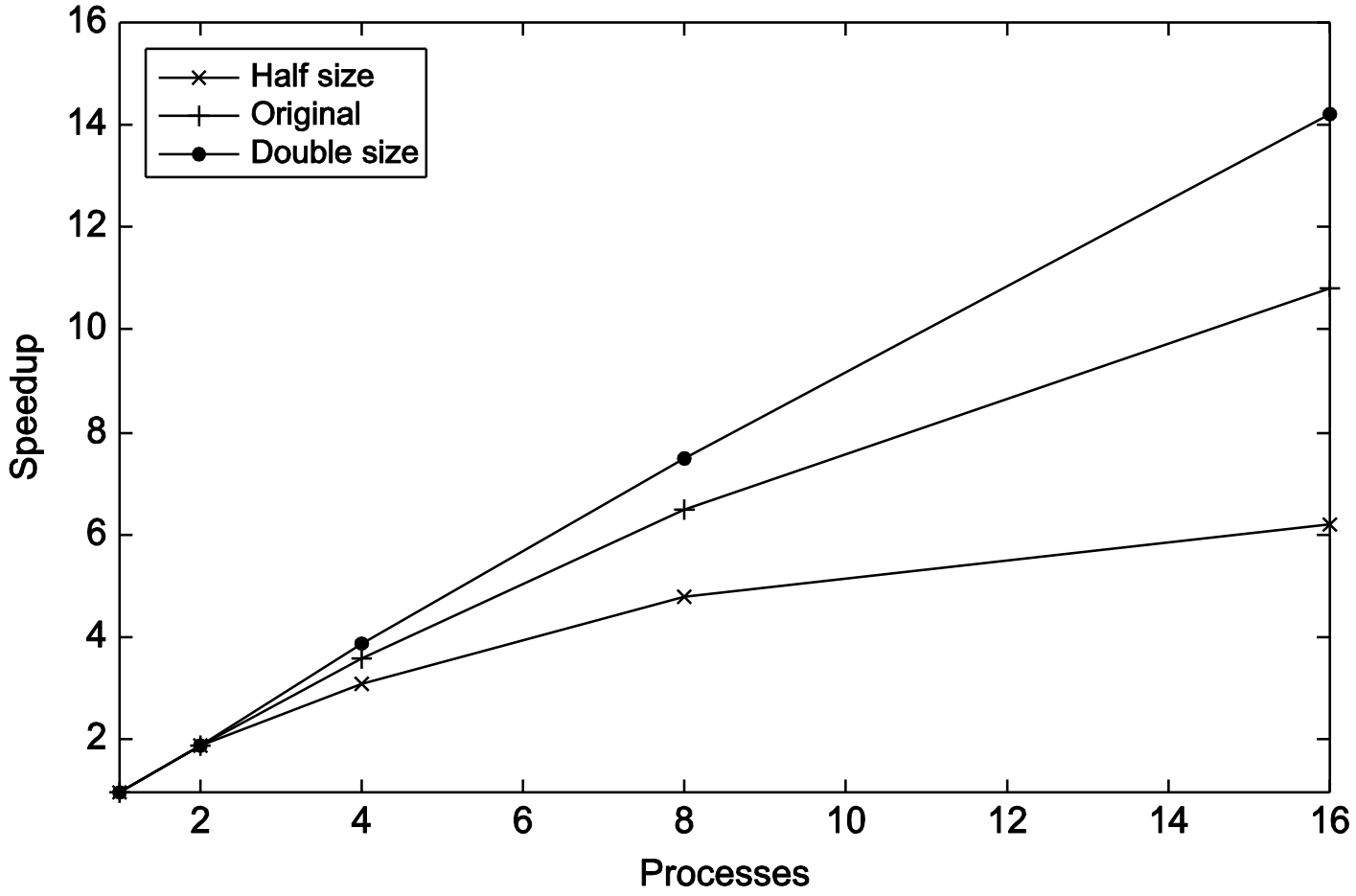
Speedups and efficiencies of a parallel program

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

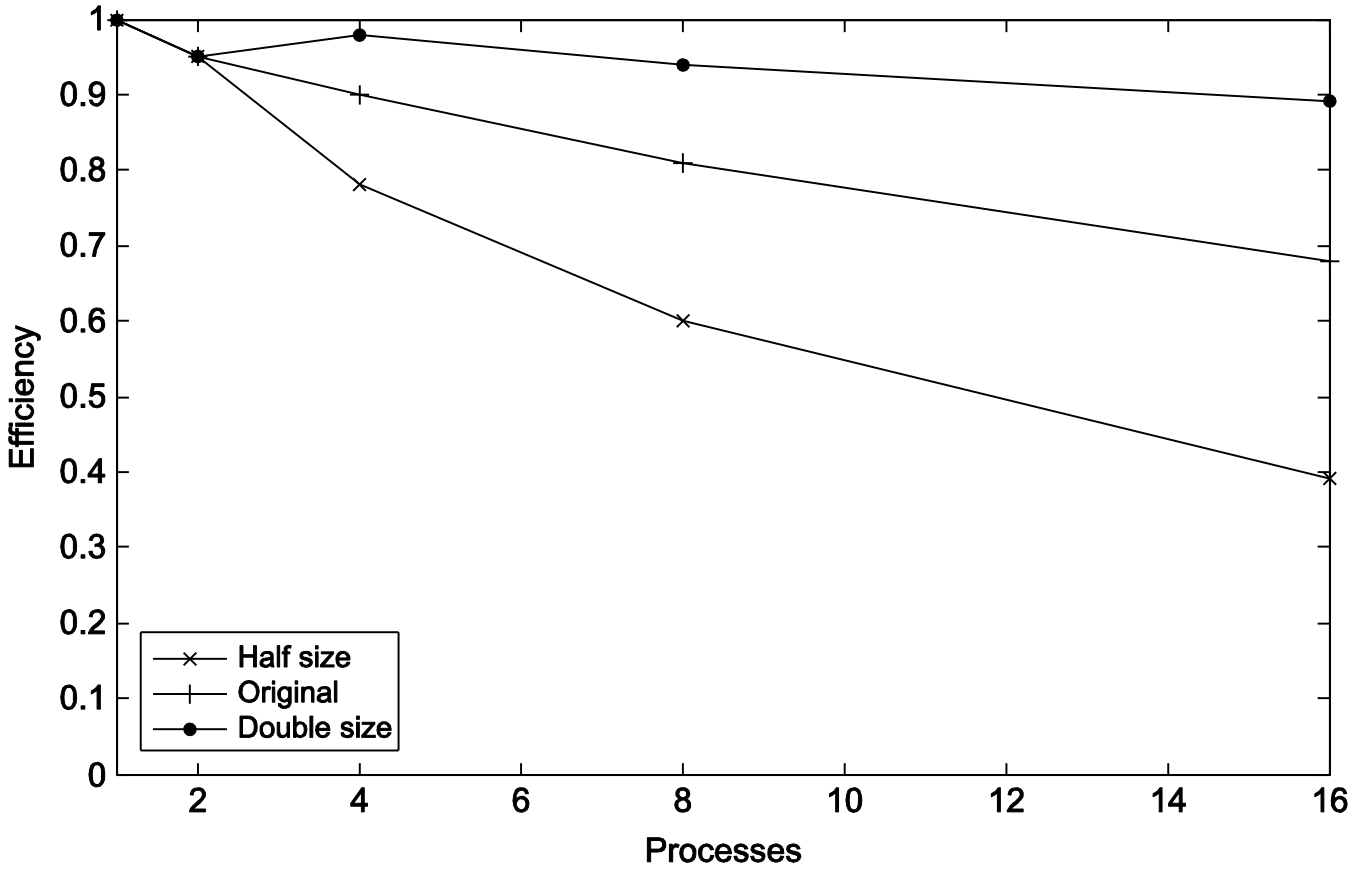
Speedups and efficiencies of parallel program on different problem sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Speedup

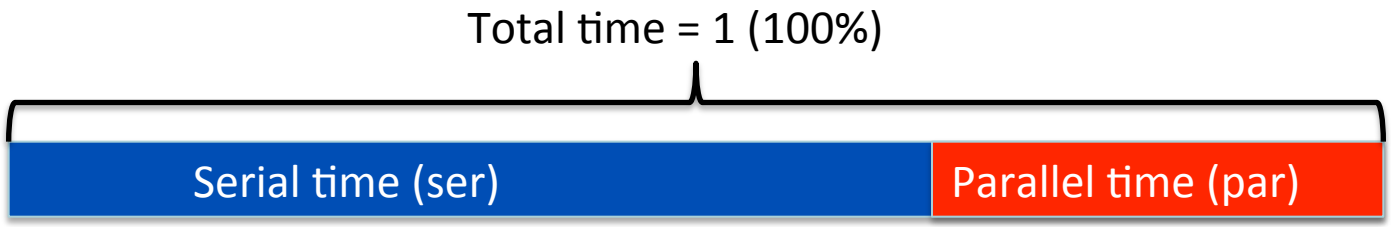


Efficiency



Amdahl's Law

- Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.



$$SpeedUp = \frac{OriginalTime}{ImprovedTime} = \frac{1}{ser + \frac{par}{\#cores}} = \frac{1}{ser + \frac{(1 - ser)}{\#cores}}$$

Scalability

- In general, a problem is *scalable* if it can handle ever increasing problem sizes.
- If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is *strongly scalable*.
- If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is *weakly scalable*.

Taking Timings

- What is time?
- Start to finish?
- A program segment of interest?
- CPU time?
- Wall clock time?

- Linux: *time <command>*

Taking Timings

- To measure time in C:
  - **#include <time.h>**
  - **struct** timeval start, stop;
  - gettimeofday(&start, **NULL**);
  - // Work of interest
  - gettimeofday(&stop, **NULL**);
  - **double** t = \

(((**double**)(stop.tv_sec)***1000.0** + (**double**)(stop.tv_usec / **1000.0**)) - \
((**double**)(start.tv_sec)***1000.0** + (**double**)(start.tv_usec / **1000.0**)));

  - fprintf(stdout, "Time elapsed = %g ms\n", t);

Other Linux commands

- Information about the hardware
  - cat /proc/cpuinfo
- Display Linux tasks (processes)
  - top
- We're are going to use GCC
  - gcc --version