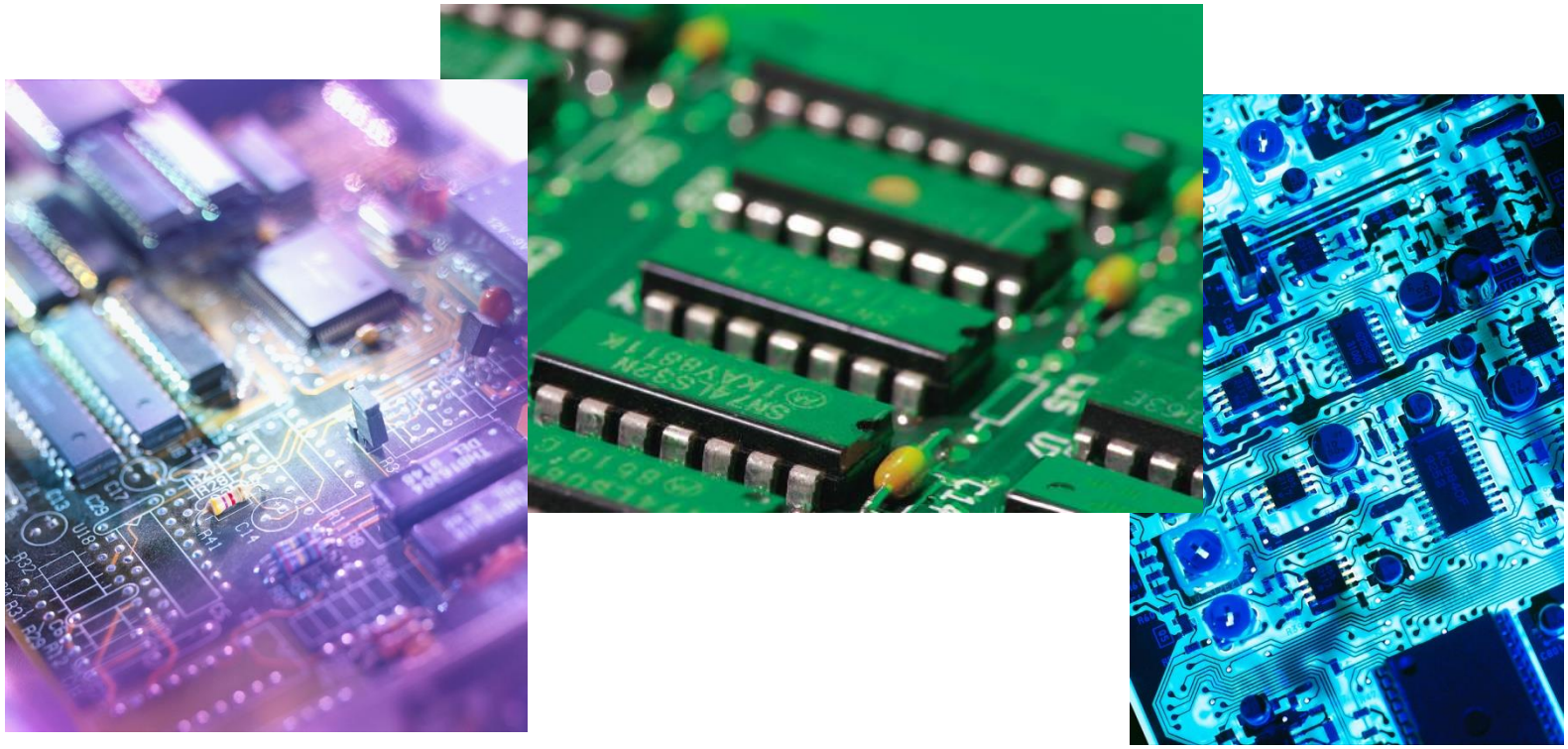
Parallel Programming

Review on Computer Hardware and Operating Systems

Slides adapted from the lecture notes by Peter Pacheco

Roadmap

- Some background
- Modifications to the von Neumann model
- Computer hardware and OS review



SOME BACKGROUND

Serial hardware and software

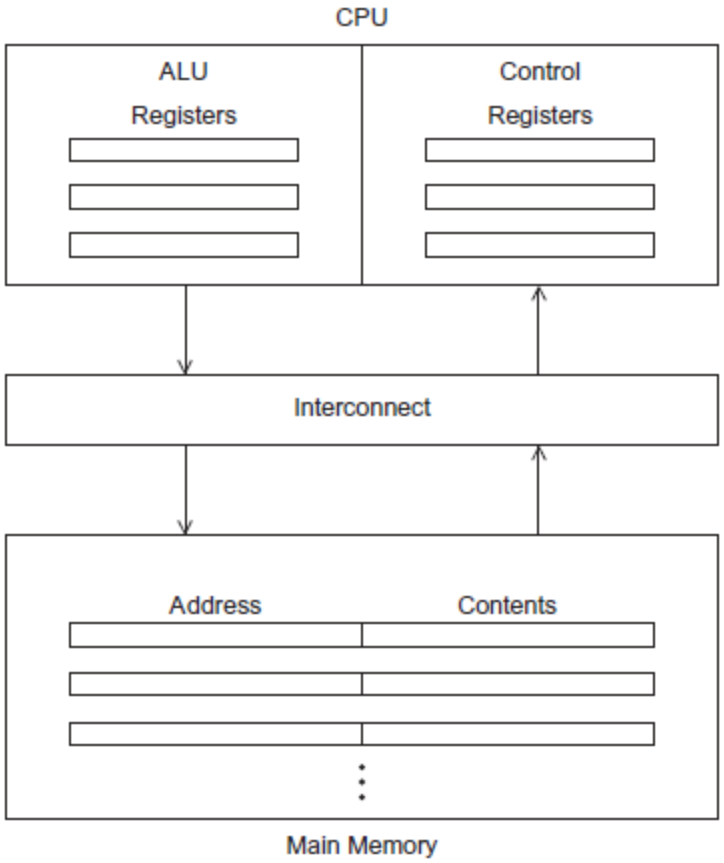
programs

input

output

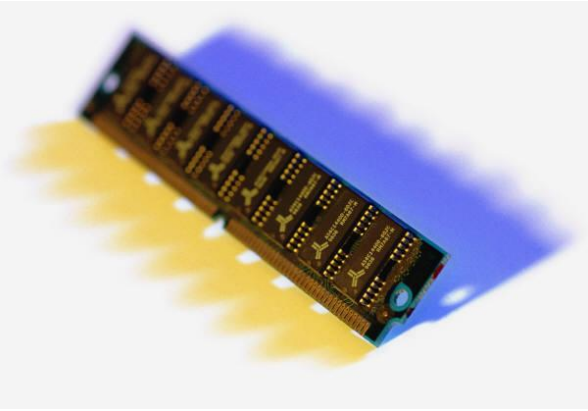
Computer runs one program at a time.

The von Neumann Architecture



Main memory

- It 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)

Two components:

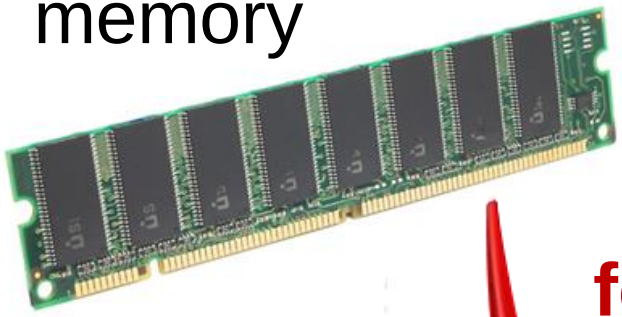
- **Control unit** - responsible for deciding which instruction in a program should be executed. (*the boss*)

- **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 that connect the CPU and memory.

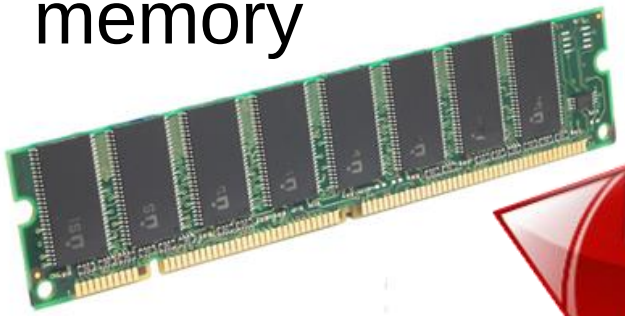
memory



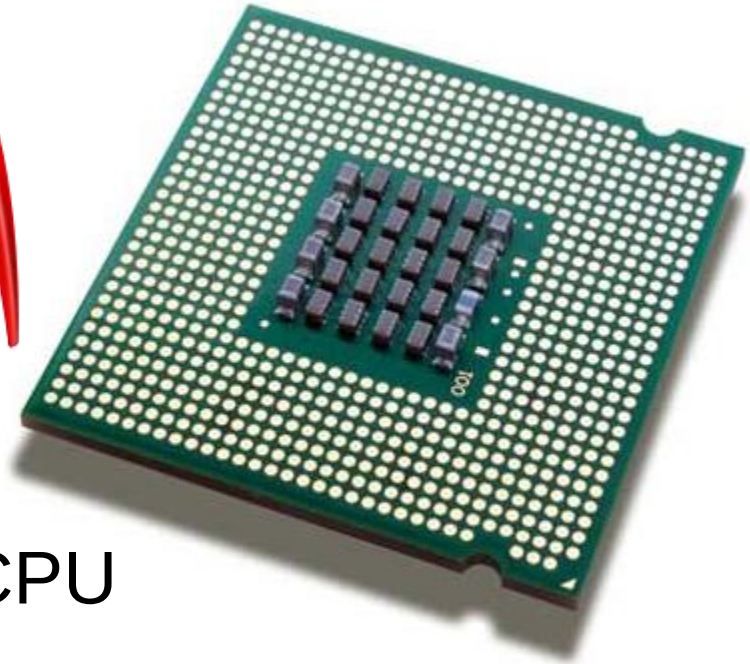
fetch/read

CPU

memory



write/store



CPU

von Neumann bottleneck

The separation of memory and CPU

- The limited storage capacity of the CPU means that large amounts of data and instructions must be transferred from the memory
- The interconnect determines the rate at which instructions and data can be accessed
- The CPU can execute instructions orders of magnitude faster than memory access

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.

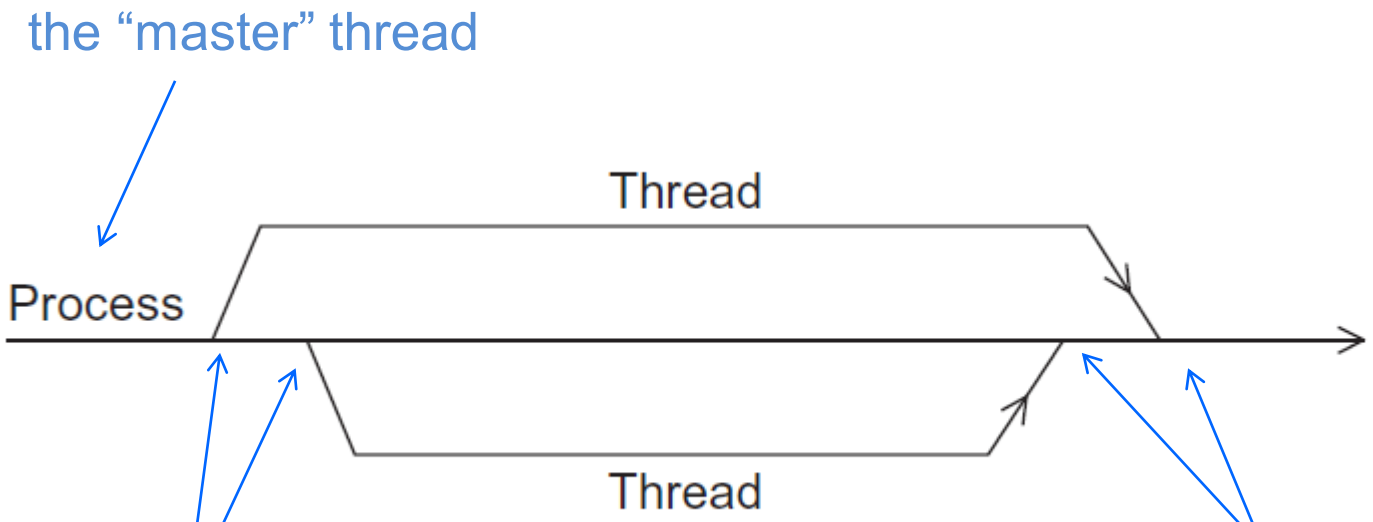
Multitasking

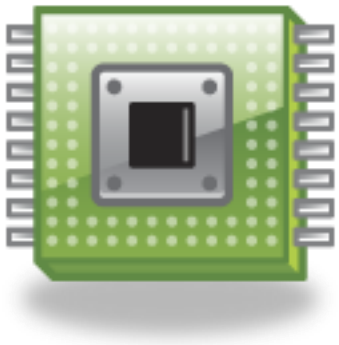
- Gives the illusion that a single processor system is running multiple programs simultaneously.
- Each process takes turns running. (**time slice**)
- After its time is up, it waits until it has a turn again. (**blocks**)

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





MODIFICATIONS TO THE VON NEUMANN MODEL

Caches

- 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.

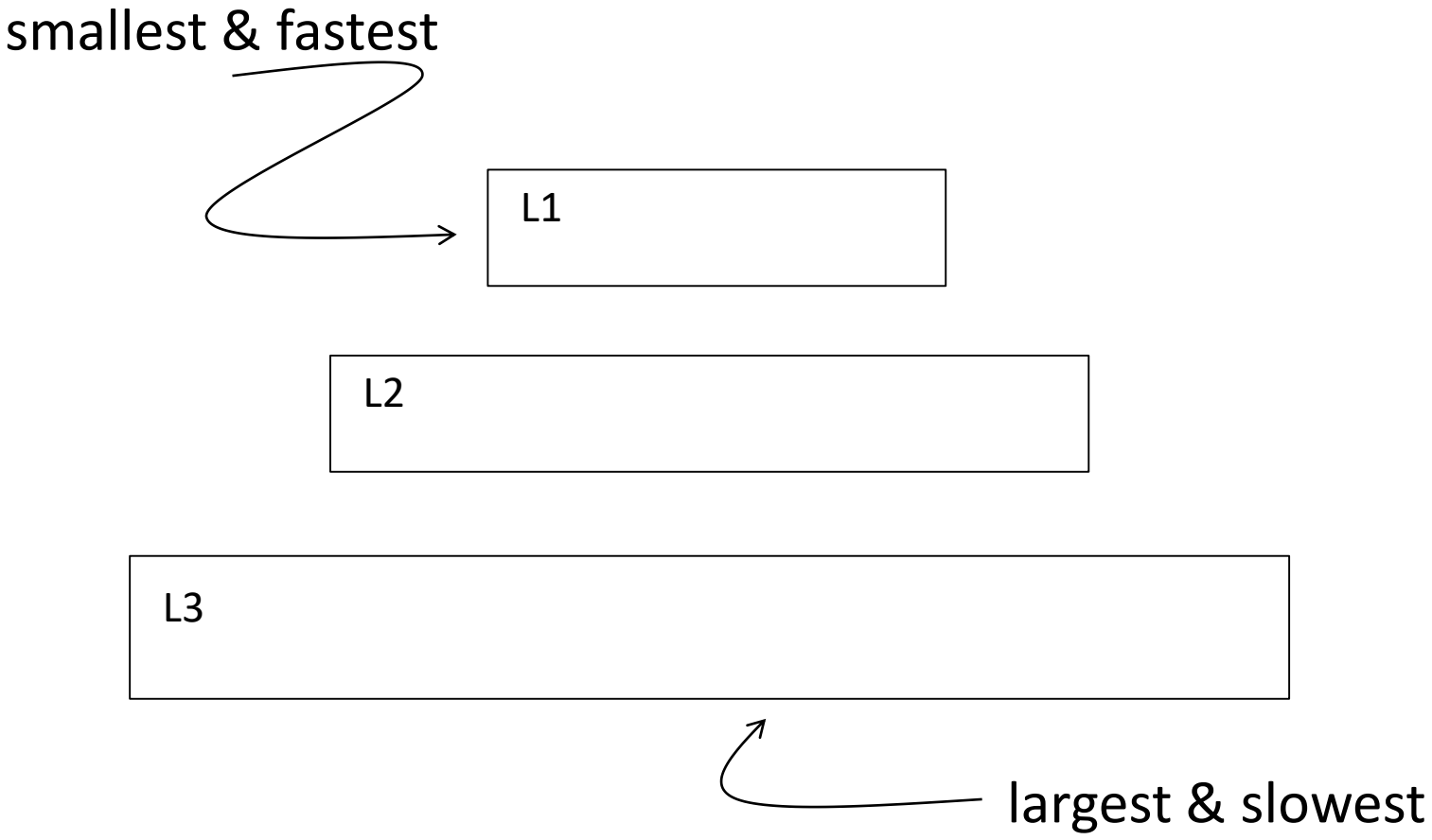
Locality

- The same or nearby locations are accessed frequently.
- **Spatial locality** – accessing a nearby location.
- **Temporal locality** – accessing in the near future.

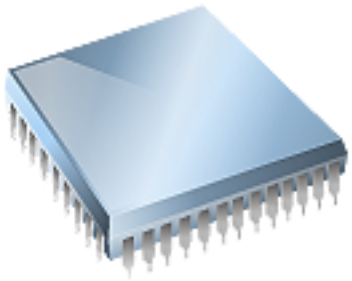
Example of locality

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

Levels of Cache



Cache hit



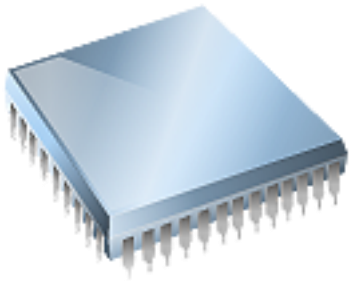
fetch x

L1 x sum

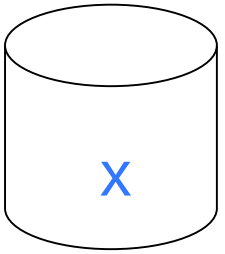
L2 y z total

L3 A[ ] radius r1 center

Cache miss



fetch x



main memory

L1 y sum

L2 r1 z total

L3 A[ ] radius center

Issues with writes

- 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 mapping

- **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.

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

Assignments of a 16-line main memory to a 4-line cache

Cache Eviction

- Caches are much smaller than main memory.
- When the cache is full, bringing a new line in memory needs to replace or evict a line in the cache.
- Common cache eviction policies include LRU/MRU (Least/Most Recently Used) and LFU (Least Frequently Used).

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.

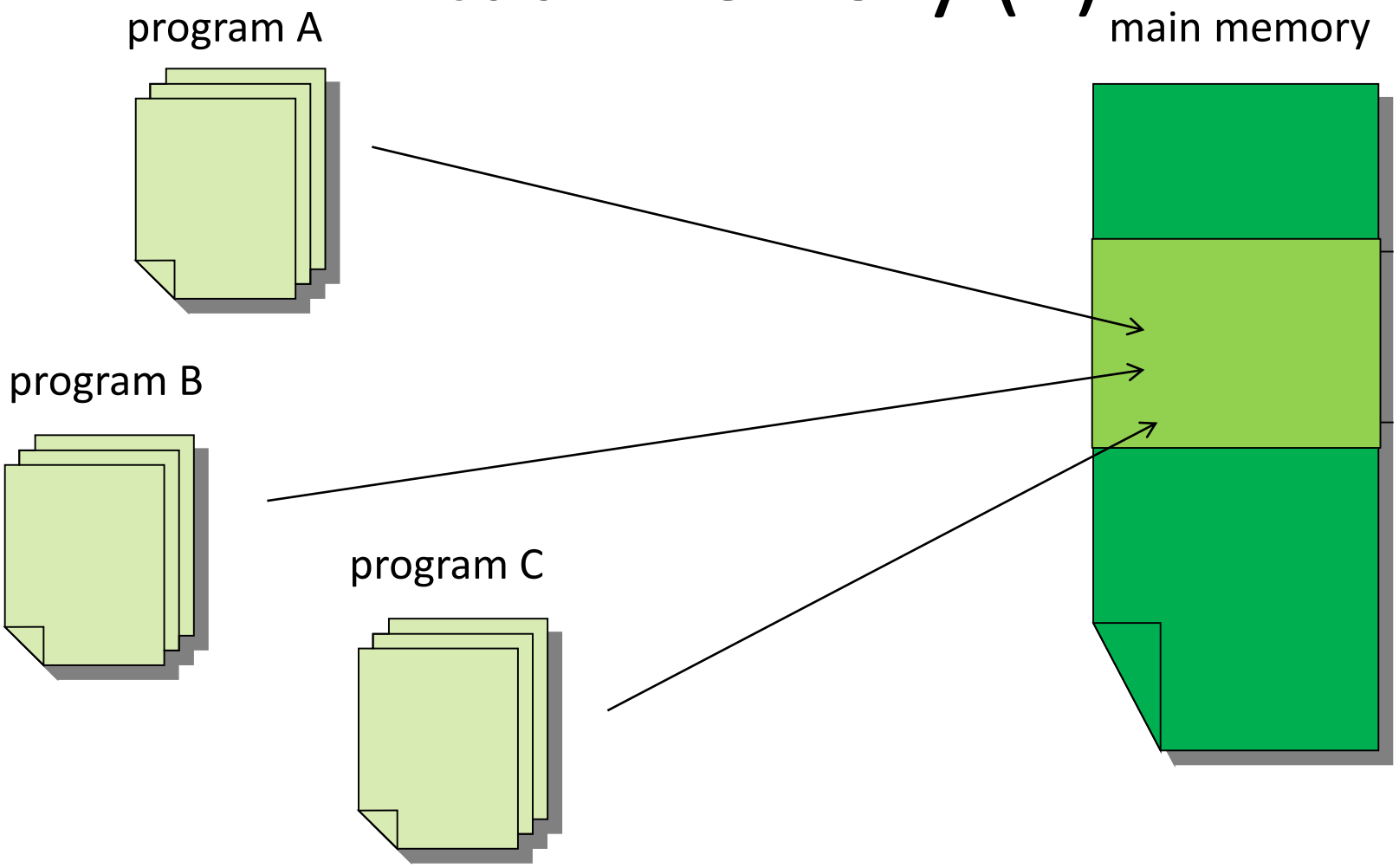
- It only keeps the active parts of running programs in main memory.

Virtual memory (3)

- **Swap space** – an area of secondary storage that keeps the inactive (parts of) running programs.

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

Virtual memory (4)



Virtual page numbers

- When a program is compiled its pages are assigned *virtual* page numbers.
- When the program is run, a table is created that maps the virtual page numbers to physical addresses.
- A **page table** is used to translate the virtual address into a physical address.

Virtual Address

Virtual Address									
Virtual Page Number					Byte Offset				
31	30	···	13	12	11	10	···	1	0
1	0	···	1	1	0	0	···	1	1

Virtual Address Divided into Virtual Page Number and Byte Offset

Translation-lookaside buffer (TLB)

- Using a page table has the potential to significantly increase each program's overall run-time.

- TLB is a special address translation cache in the processor.

Translation-lookaside buffer (2)

- It caches a small number of entries (typically 16–512) from the page table in very fast memory.

- **Page fault** – attempting to access a valid physical address for a page in the page table but the page is only stored on disk.

Instruction Level Parallelism (ILP)

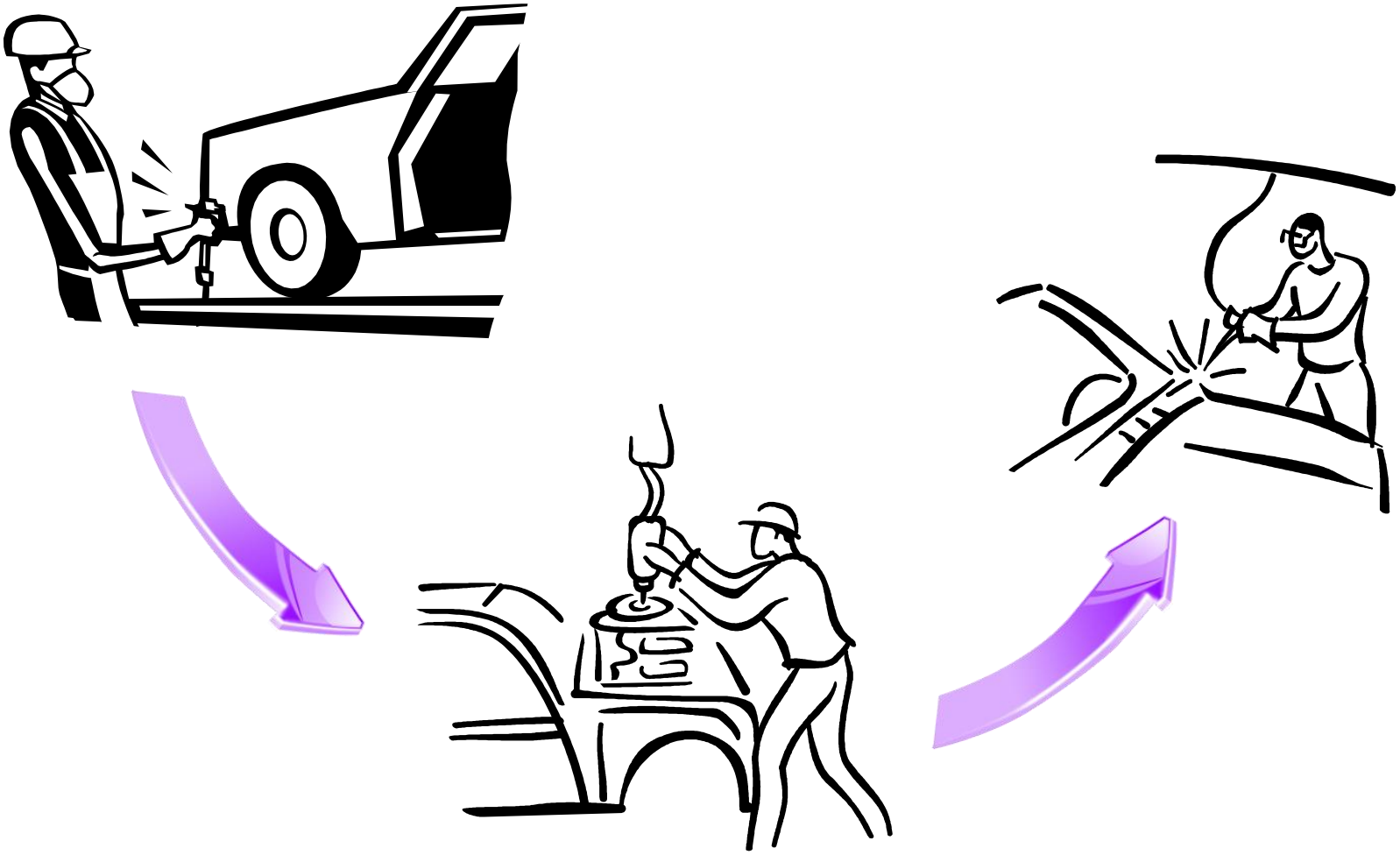
- Attempts to improve processor performance by having multiple processor components or **functional units** simultaneously executing instructions.

Instruction Level Parallelism (2)

- **Pipelining** - functional units are arranged in stages.

- **Multiple issue** - multiple instructions can be simultaneously initiated.

Pipelining



Pipelining example (1)

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Add the floating point numbers
9.87×10^4 and 6.54×10^3

Pipelining example (2)

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

- Assume each operation takes one nanosecond (10^{-9} seconds).
- This for loop takes about 7000 nanoseconds.

Pipelining (3)

- Divide the floating point adder into 7 separate pieces of hardware or functional units.
- First unit fetches two operands, second unit compares exponents, etc.
- Output of one functional unit is input to the next.

Pipelining (4)

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Pipelined Addition.

Numbers in the table are subscripts of operands/results.

Pipelining (5)

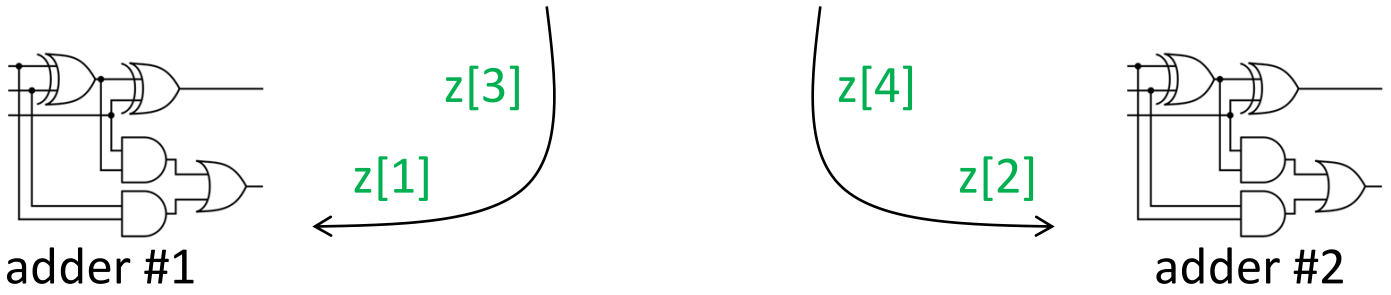
- One floating point addition still takes 7 nanoseconds.

- But 1000 floating point additions now takes 1006 nanoseconds!

Multiple Issue (1)

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

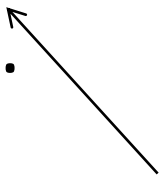
```
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```



Multiple Issue (2)

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

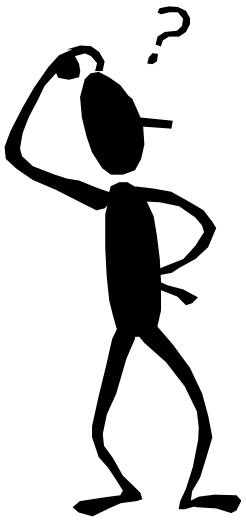
- **dynamic** multiple issue – functional units are scheduled at run-time.



superscalar

Speculation (1)

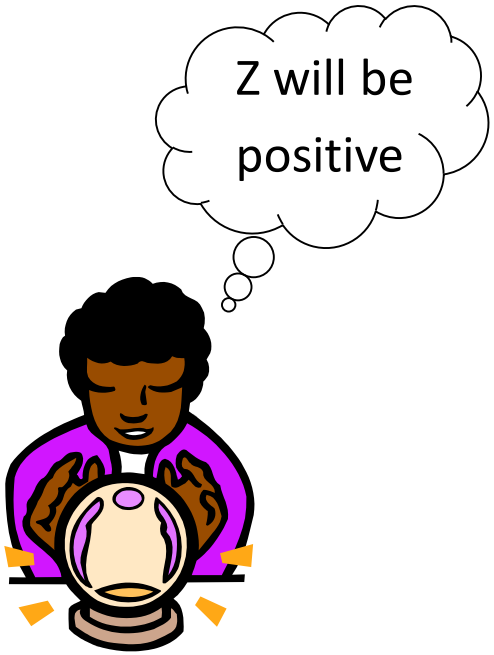
- In order to make use of multiple issue, the system must find instructions that can be executed simultaneously.



  - In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess.

Speculation (2)

```
z = x + y ;
i f ( z > 0)
    w = x ;
e l s e
    w = y ;
```



If the system speculates incorrectly,
it must go back and recalculate w = y.

Hardware multithreading (1)

- There aren't always good opportunities for simultaneous execution of different threads.
- Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled.
  - Ex., the current task has to wait for data to be loaded from memory.

Hardware multithreading (2)

- **Fine-grained** - the processor switches between threads after each instruction, skipping threads that are stalled.

  - Pros: potential to avoid wasted machine time due to stalls.
  - Cons: a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction.

Hardware multithreading (3)

- **Coarse-grained** - only switches threads that are stalled waiting for a time-consuming operation to complete.

  - Pros: switching threads doesn't need to be nearly instantaneous.
  - Cons: the processor can be idled on shorter stalls, and thread switching will also cause delays.

Hardware multithreading (3)

- **Simultaneous multithreading** (SMT) - a variation on fine-grained multithreading.

- Allows multiple threads to make use of the multiple functional units.