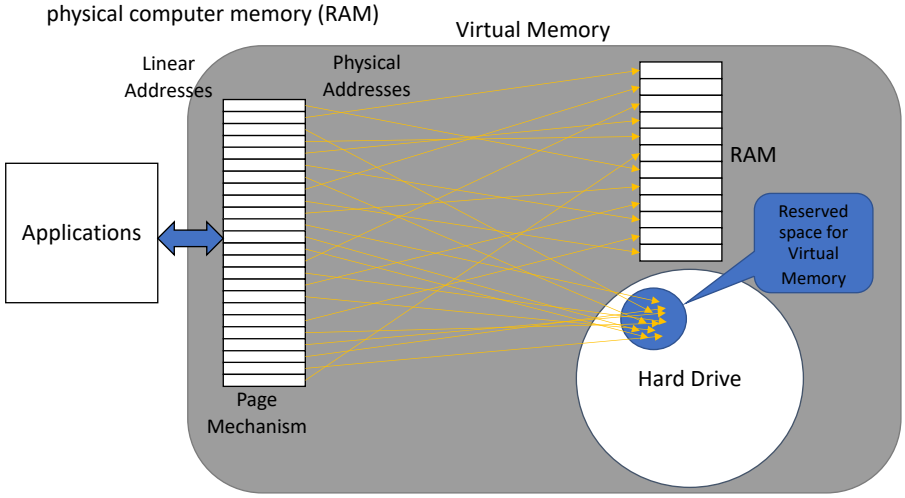
Virtual Memory
Cache
Techniques to improve performance

1

Virtual Memory

Using Virtual Memory, the applications can have access to more memory than the physical computer memory (RAM)



2

Virtual Memory

When an application needs memory, for example 100MB, the Operating System Reserves a **continuous** space in virtual addresses

The application receive from the Operating System a pointer to the first byte

Using an index added to the pointer, the application can use all the 100MB of memory

The application see this memory as a physical memory

The address never change along the program execution

The address space is divided in 4KB blocks

Each block can be mapped in the RAM or in the hard drive

The application doesn't have, at the same time, all the blocks in RAM

3

Virtual Memory

When the application points to a block that is mapped in the hard drive, the system reads this block from the hard drive and write it in any free block in the physical RAM

Just after that, the application can use the data

This is the reason why the computer diminish the applications performance when use Virtual Memory

If no block in RAM is free, the system transfers, at least, one block of RAM to the hard drive

To improve the performance, the system maintain a large number of blocks free in RAM

This is important to get free space whenever an application needs more memory, and the system can offer this memory immediately.

4

Virtual Memory

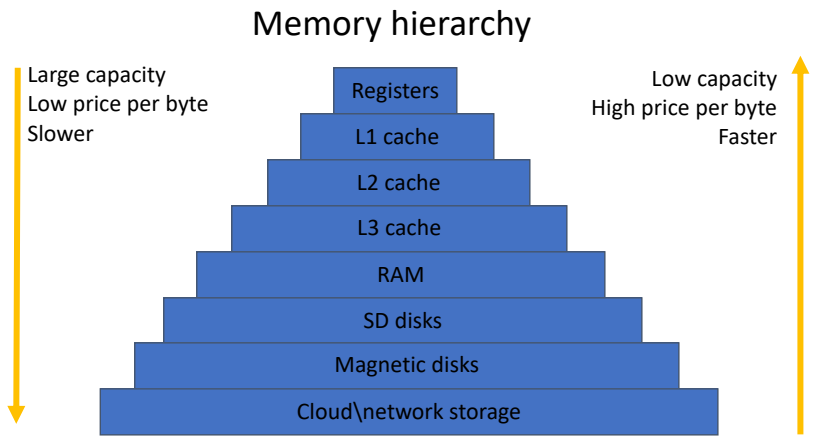
The system select the blocks in RAM that aren't used at a long time to free, putting it contends in the hard drive

The system maintains a *paging mechanism* that maps all the blocks in RAM or in the hard drive

The physical position of any block can change along the program execution, but this change is completely transparent to the applications, since they don't know where is, physically, the used memory

5

Cache memory



6

Cache memory

Principle 90/10

It's a principle, not a precise rule…

In general, 90% of the time that a program use, is use just by 10% of the code

This means that are assembly lines more used than others

In general, 10% of the variables that a program use, represent 90% of the total time that a program use for all variables

This means that are variables more used than others

7

Principle 90/10

- **Some code is executed much more often than other code.** For example, some error handling code might never be used. Some code will be executed only when you start your program. Other code will be executed over and over while your program runs.

- **Some code takes *much* longer to run than other code.** For example, a single line that runs a query on a database or pulls a file from the internet will probably take longer than millions of mathematical operations.

8

Principle 90/10

The point is, **if you need your program to run faster, probably only a small number of lines is significant to making that happen.**

9

Cache memory

Principle of continually

If a program access a specific byte in an address at a given time, there is high probability that in near time the program access the near addresses bytes.

10

Cache memory

Principle of temporality

If a program access a specific byte at a given time, there is high probability that in near time the program access again to this byte.

11

Why use cache memory?

To speedup the computer

This is possible if the memory cache can reduce the CPU time to get any data from memory

Imagine a worker riding wheels on an automobile assembly line

If he does not have the wheels and bolts around, he must spend a lot of time to get them from the warehouse

In this case he spends more time to make her job

12

Cache Operation

- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache –> (fast)
- If not, read required block from main memory to cache -> (slow)
- Then deliver from cache to CPU

13

What is a block?

The cache memory is divided in blocks with the same size

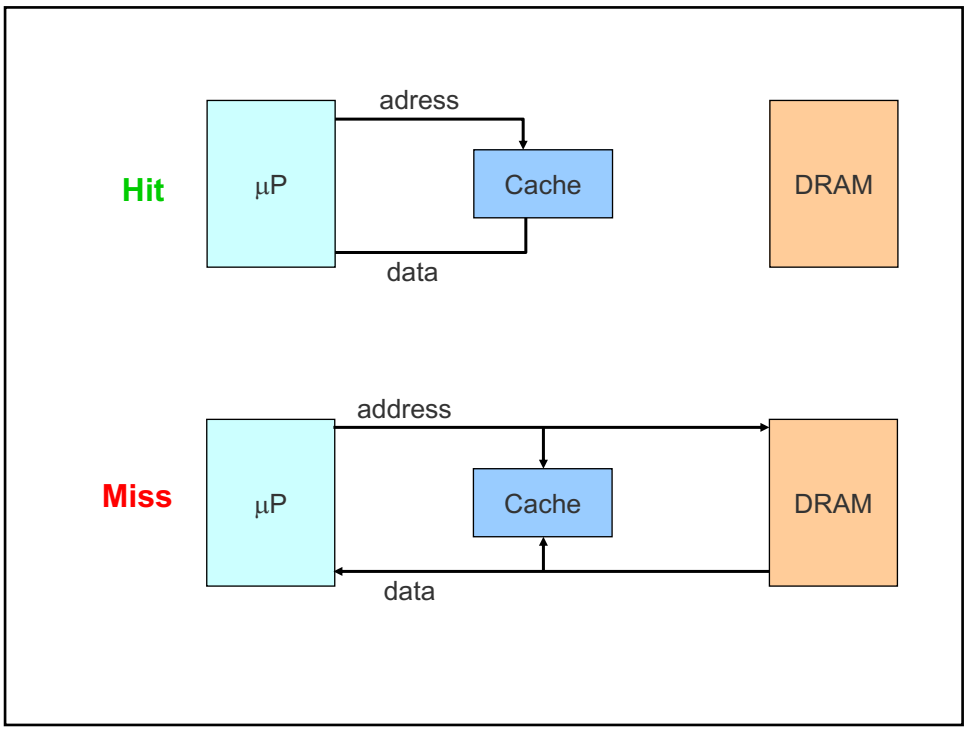
Whenever the CPU needs a specific byte and this byte isn't in the cache, the system get from memory an entire block of data, instead a single byte

In this case we have a "**cash miss**" and an augment of time need to get the entire block from memory

According the principals of continually and temporality is high probable that in next times the CPU access to the same byte or the nears bytes

In this case we have a "**cash hit**" and the time to CPU get the data is very low

14



15

What is a block?

Imagine a large box divide in small boxes to store screws

The large box is the cache, and the small boxes are the blocks

Whenever the worker needs a specific screw that isn't in the large box, he goes to warehouse and take an entire small box of the needed screws

In the next time that he needs the same screw, theses screws are in the box

16

What is the best size for the blocks?

If the block size grows up

- The number of blocks in the cache diminish

  (number of blocks=cache size/block size)

- Diminish the number of blocks diminish the variety of data
  (if we have only one block in the cache, we just have one type of "screws")
- Having low variety of data, grows up the number of cache misses
- It needs more time to transfer the entire block from memory to the cache

17

What is the best size for the blocks?

If the block size decreases

- The number of blocks in the cache increase and increase the variety of data
- The number of bytes in the block diminish (if the size is just one byte whenever we need another byte, we must get it from memory having always a cache miss)
- **This isn't efficient**

18

What is the best size for the blocks?

We must find compromises…

The most used block size is 64 bytes and 128 bytes

The block size is always a power of 2

In general, when the cache size is higher, the block size is also higher

19

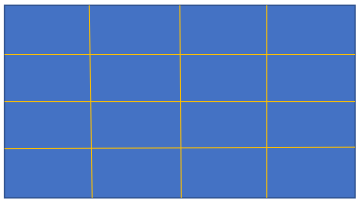
Where put each block?

Suppose again we have a large box with 16 blocks to put screws

You can organize the box by screws size

Each block have only one screw size at a time

At different times each block can have different screw size



20

Where put each block?

For example:

block 0 can have screws size of 1, 17, 33, 49, …

block 1 can have screws size of 2, 18, 34, 50, …

block 2 can have screws size of 3, 19, 35, 51, …

…

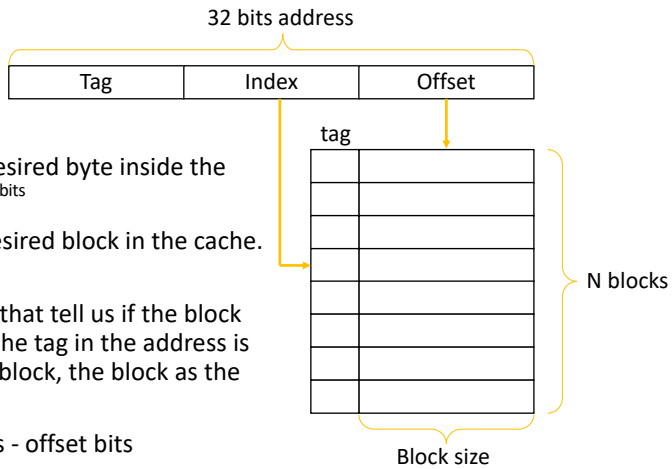
block 15 can have screws size of 16, 32, 48, 64, …

If we need the size 50 screw, we know that this screw only can be in the block 1. We just need to confirm if the block 1 contains the size 50 screw.

If this block does not have the size 50 screw, we must empty that block and put there the size 50 screws

21

Direct mapping



The offset select the desired byte inside the block. Block size=$2^{\text{offset bits}}$

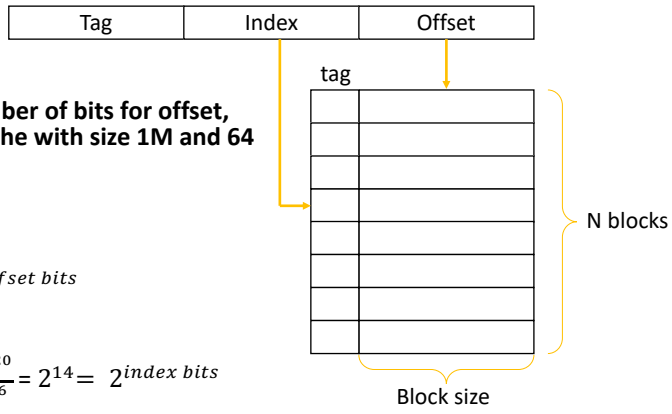
The index select the desired block in the cache. N blocks=$2^{\text{index bits}}$

The tag is an identifier that tell us if the block as the desired data. If the tag in the address is equal to the tag in the block, the block as the desired data.

Tag bits =32 - index bits - offset bits

22

Direct mapping



Exercise: Find the number of bits for offset, index and tag for a cache with size 1M and 64 bytes of block size

Cache size=1M=2^{20}

Block size=64=2^6 = $2^{offset\ bits}$

Offset bits=6

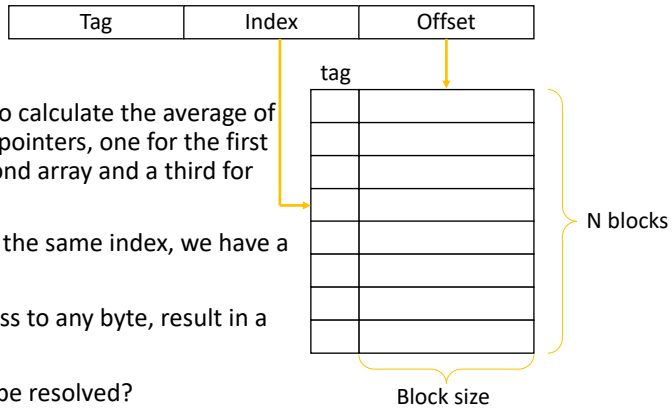
N blocks=$\frac{Cache\ size}{Block\ size}$ = $\frac{2^{20}}{2^6}$ = 2^{14}= $2^{index\ bits}$

Index bits=14

Tag bits=32-index bits-offset bits=32-14-6=12

23

Direct mapping problems



Consider the exercise to calculate the average of two arrays. We have 3 pointers, one for the first array, other for de second array and a third for the result.

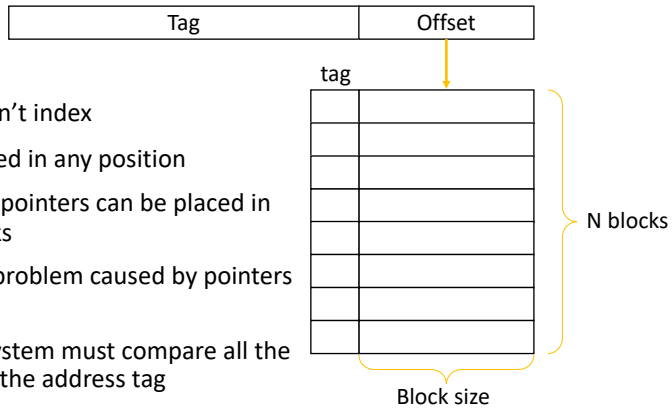
If all this pointers have the same index, we have a huge problem!

In this case, every access to any byte, result in a cache miss.

How can this problem be resolved?

24

Associative mapping (no index)



In this method there isn't index

Each block can be placed in any position

Now any two different pointers can be placed in any two different blocks

This solve the conflict problem caused by pointers with the same index

To find the block the system must compare all the tags in the blocks with the address tag

This can take a long time and isn't efficient

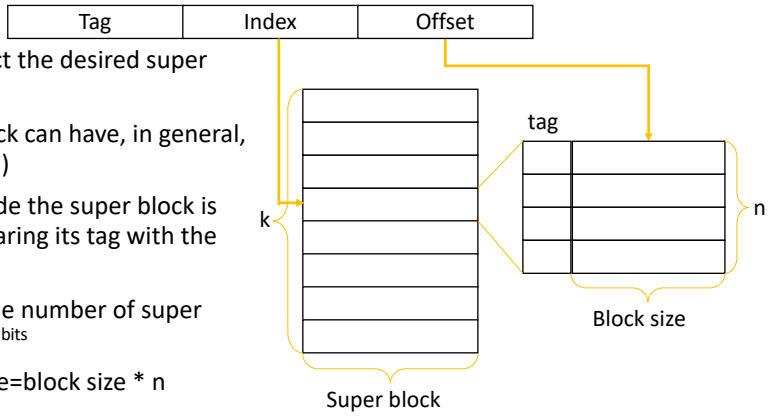
25

Memory cache

How can we find a new mapping method for the cache that has the advantages of direct mapping and associative mapping, but does not have its disadvantages?

26

Associative mapping by blocks



The index select the desired super block

Each super block can have, in general, 4 or 8 blocks (n)

Each block inside the super block is selected comparing its tag with the tag in address

K represents the number of super blocks. $K=2^{\text{index bits}}$

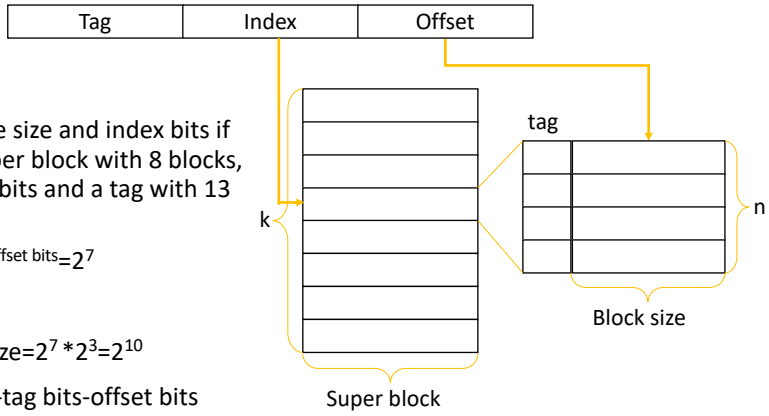
Super block size=block size * n

Cache size=super block size * k

Block size=$2^{\text{offset bits}}$

27

Associative mapping by blocks



Exercise

Find the cache size and index bits if we have a super block with 8 blocks, an offset of 7 bits and a tag with 13 bits

Block size =$2^{\text{offset bits}}=2^7$

n=8=2^3

Super block size=$2^7 * 2^3=2^{10}$

Index bits=32-tag bits-offset bits

Index bits=32-13-7=12

Cache size=$2^{12}*2^{10}=2^{22}$=4MB

28

How process writes in cache?

In general programs make more reads in memory that writes

The cache can have different politics for read and for write

For write its possible implement the following politics:

- Write directly in memory
  - If the block is in cache write also in the block
  - If the block is not in cache load and write it
  - If the block in not in cache don’t load it

  In these modes when a block must be changed isn't necessaire copy it contends to the ram
- Write only in the block
  - If the block is not in cache load and write it

  In this mode when a block must be changed is necessaire copy it contends to the ram if they have been changed

29

Which block to replace?

In direct mapping

- The block pointed by index (the only possibility)

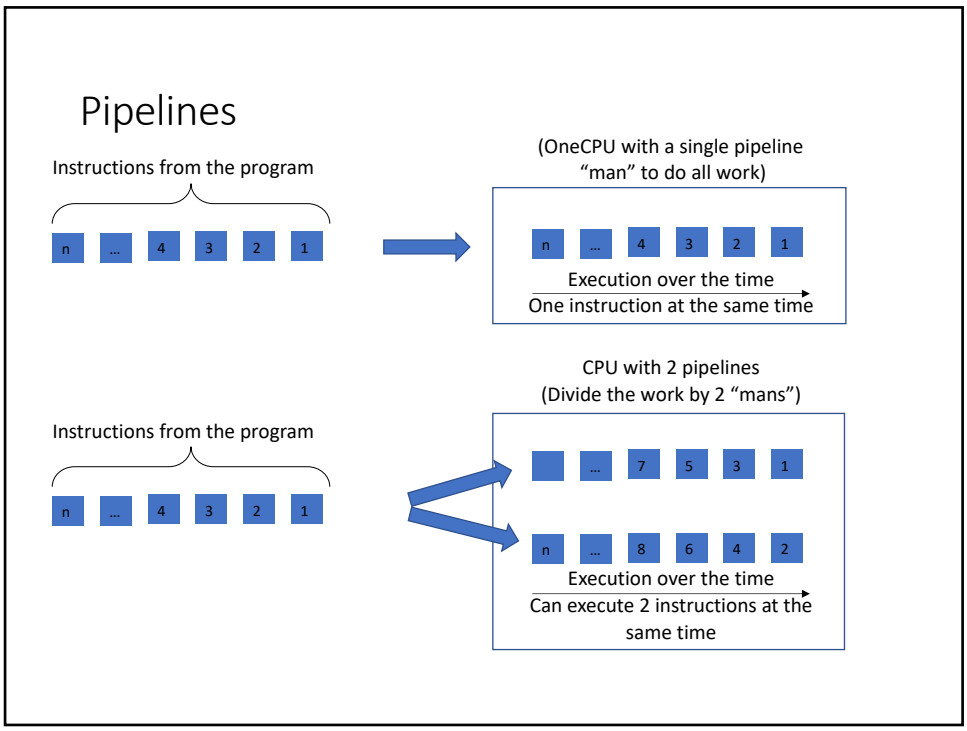
In associative mapping

- The block that has not been used for a long time
- Random block

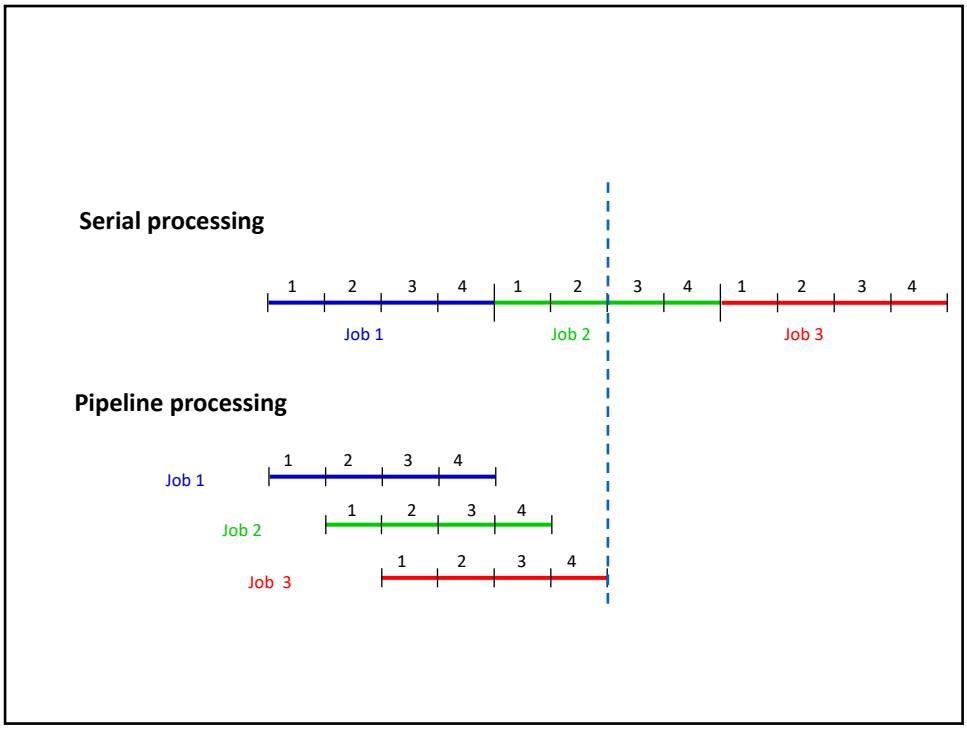
In associative mapping by blocks

- Inside the super block
  - The block that has not been used for a long time
  - Random block

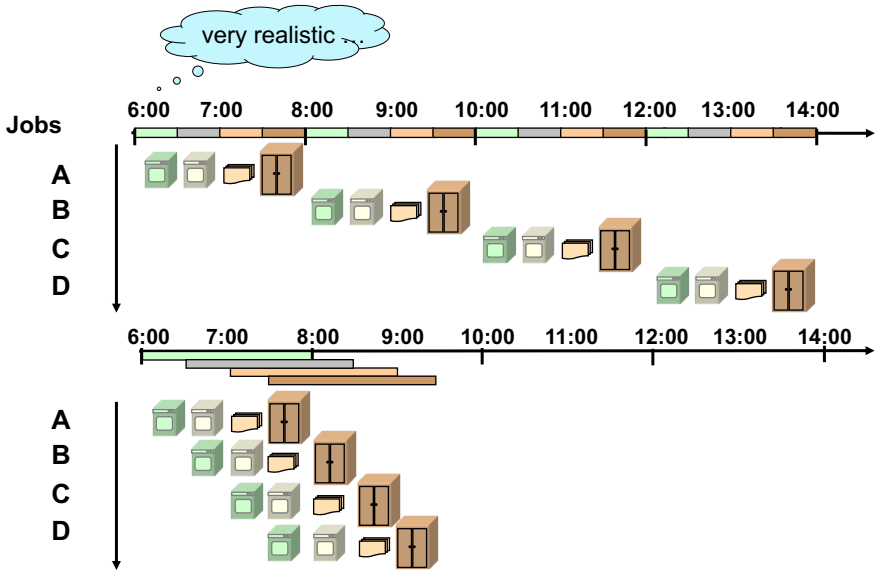
30


Pipelines
Instructions from the program
n
...
4
3
2
1
(OneCPU with a single pipeline
"man" to do all work)
Execution over the time
One instruction at the same time
CPU with 2 pipelines
(Divide the work by 2 "mans")
7
5
3
8
6
4
Can execute 2 instructions at the same time

31


Serial processing
Job 1
Job 2
Job 3
Pipeline processing
Job 1
Job 2
Job 3

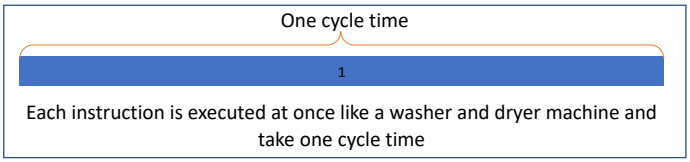
32



33

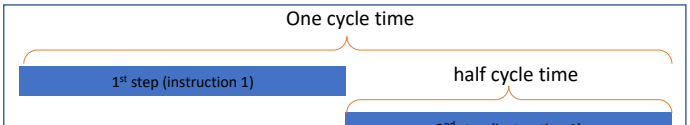
States in pipelines

One state pipeline



Each instruction is executed at once like a washer and dryer machine and take one cycle time

Two states pipeline

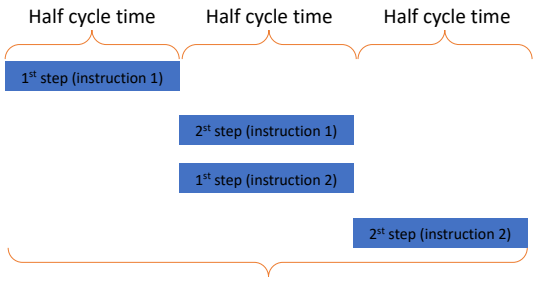


Each instruction is divided in two steps. Each step is executed in a dedicated and independent hardware like one washer machine to wash and other dryer machine to dry.
The total time for the instruction execution is de same (or approximated).
No benefits for a single instruction!

34

States in pipelines

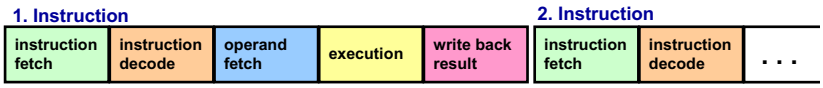
Two states pipeline, two instructions



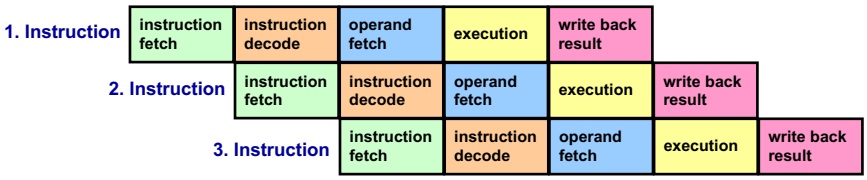
1.5 cycle time
25% faster than one state pipeline for 2 instructions
Tends to 50% faster when the number of instructions increase

35

Sequential execution:



Pipelining:



36

Techniques to improve performance
Branch prediction\speculative execution

```
next:   mov [esi],al
        inc al
        inc esi
        dec ecx
        jnz next
        mov al,'A'
```

The instruction *jnz next* blocks all successive instruction because just after this instruction is finish the CPU knows the next instruction

If the CPU can predict how a branch is made, it can initialize the execution of the next instruction

If the prediction is confirmed the CPU gains time

If not, the CPU suspend the instruction execution and execute the correct instruction

37

Predictions

- Pro
  - Able to eliminate a branch and therefore the associated branch prediction ➡ increasing the distance between mispredictions.
  - The run length of a code block is increased ➡ better compiler scheduling.
- Contra
  - Predication affects the instruction set, adds a port to the register file, and complicates instruction execution.
  - Predicated instructions that are discarded still consume processor resources; especially the fetch bandwidth.
- Predication is most effective when control dependencies can be eliminated, such as in an `if-then` with a small `then` body.
- The use of predicated instructions is limited when the control flow involves more than a simple alternative sequence.

38

Techniques to improve performance
Out of order execution

```
mov al,bl
add al,cl
mov ah,'A'
```

The CPU can change the execution order to

```
mov al,bl
mov ah,'A'
add al,cl
```

Changing the order, the CPU can execute these 3 instructions in less time since the instruction mov ah,'A' can be executed in parallel with the instruction mov al,bl

39

Techniques to improve performance

Registers rename

```
add eax,5
add ebx,eax
mov eax,2
```

These 3 instructions have dependency between then.
The second just can be executed after the first and the third just can be executed after the second.
CPU can solve this dependency internally renaming the register eax.

40

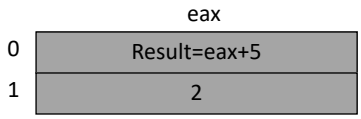
Techniques to improve performance
Registers rename

Each register have internally a double box
At a time only one box (0 or 1) is active

```
add eax,5;      if the active box is 0 put the result in box '0
add ebx,eax;    use the value in the active box (0)
mov eax,2;      change the active box to 1 and put in it the value 2
```

This instruction can be executed in parallel with the second

This process is completely independent for the programmer



41

Techniques to improve performance

1. **Branch prediction\speculative execution**
2. **Out of order execution**
3. **Registers rename**

42