Compilers and computer architecture

16. Memory layout and locality in code generation and hardware

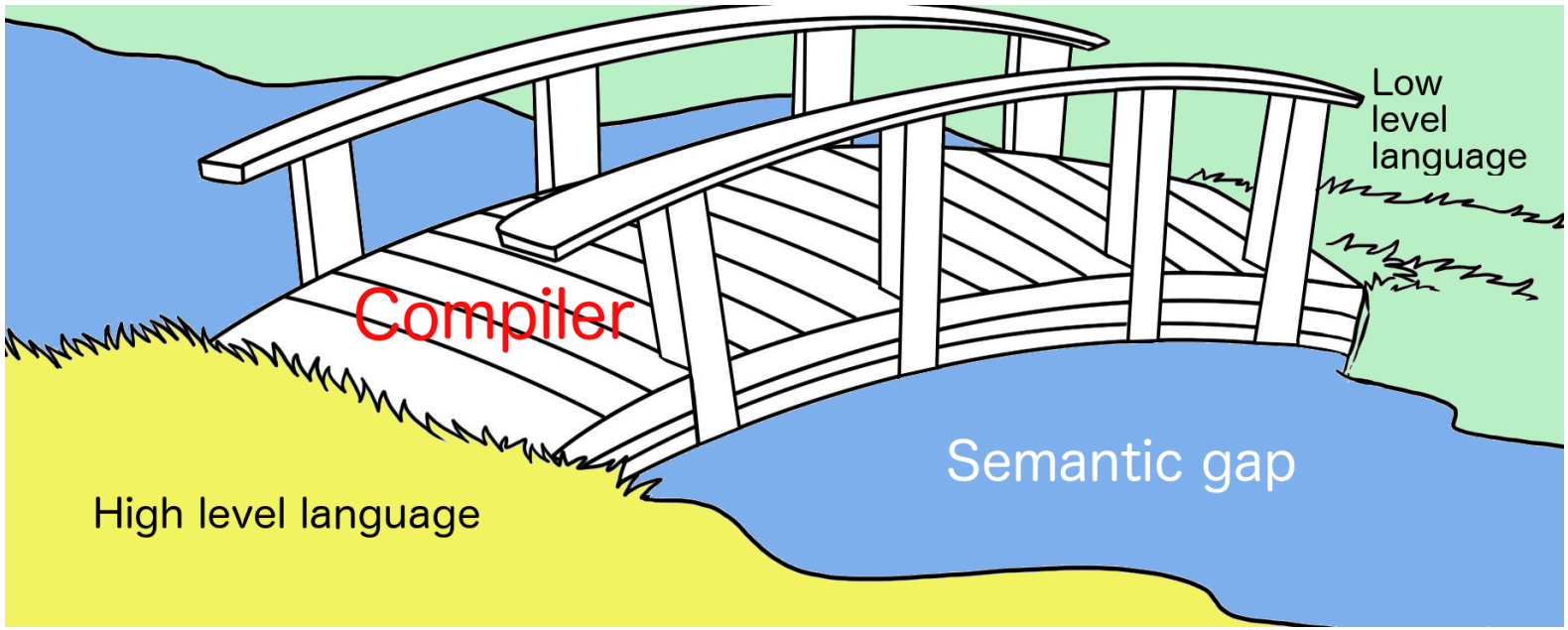
Hsi-Ming Ho

(slides by Martin Berger)

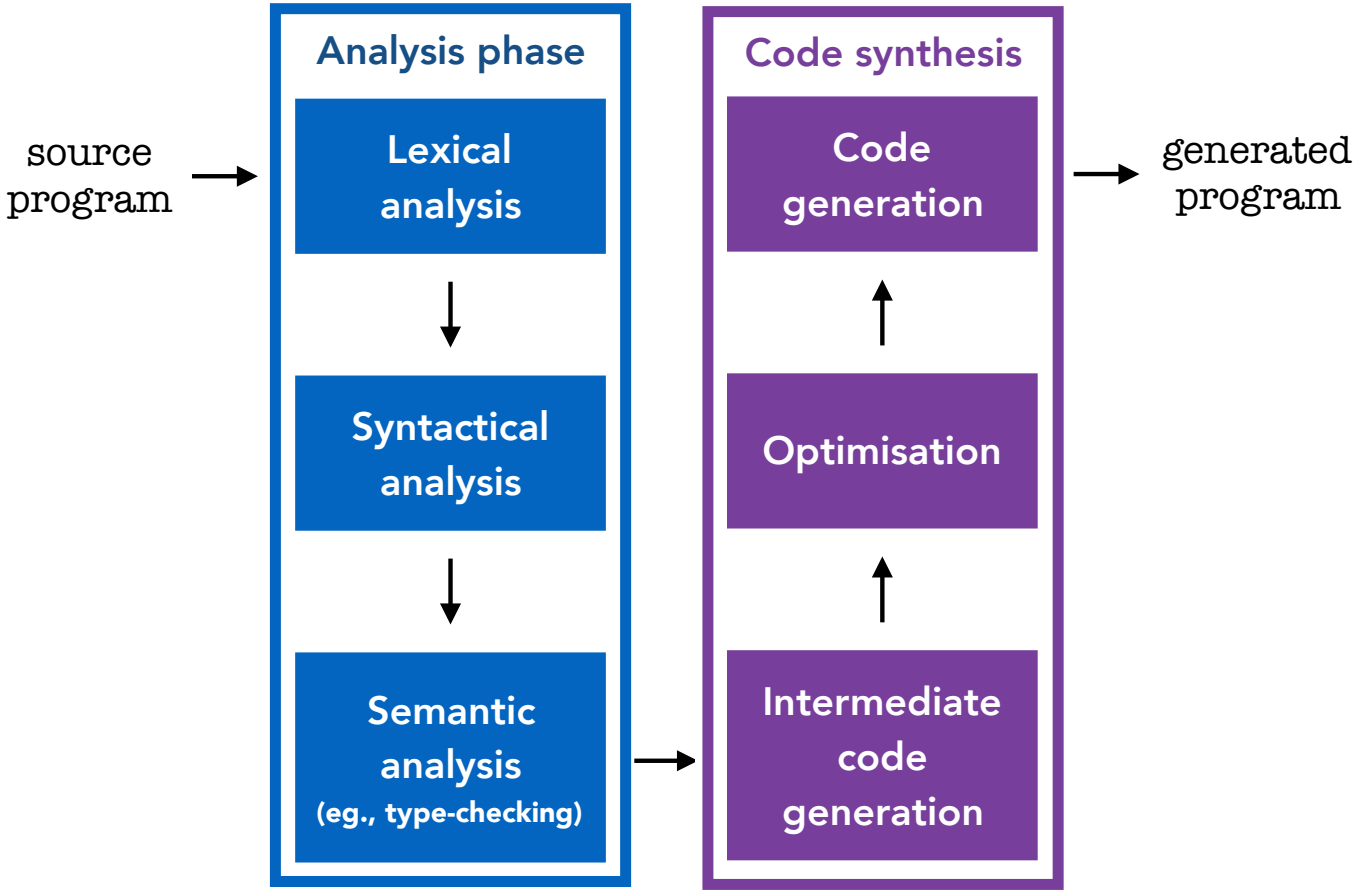
Email: **hsi-ming.ho@sussex.ac.uk**
Office hours: Online, by email appointment

Autumn 2022

Recall the function of compilers

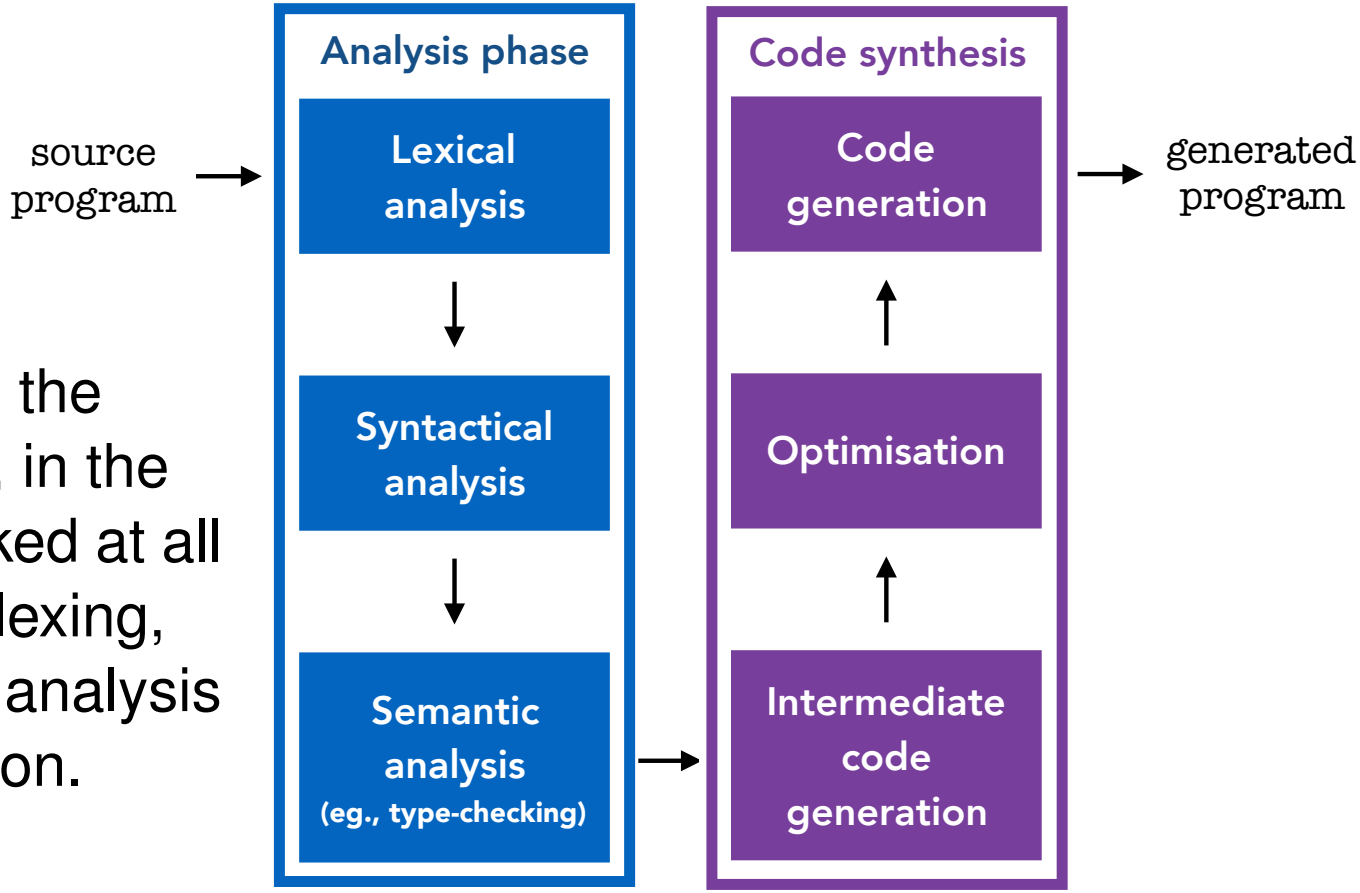


Where we are

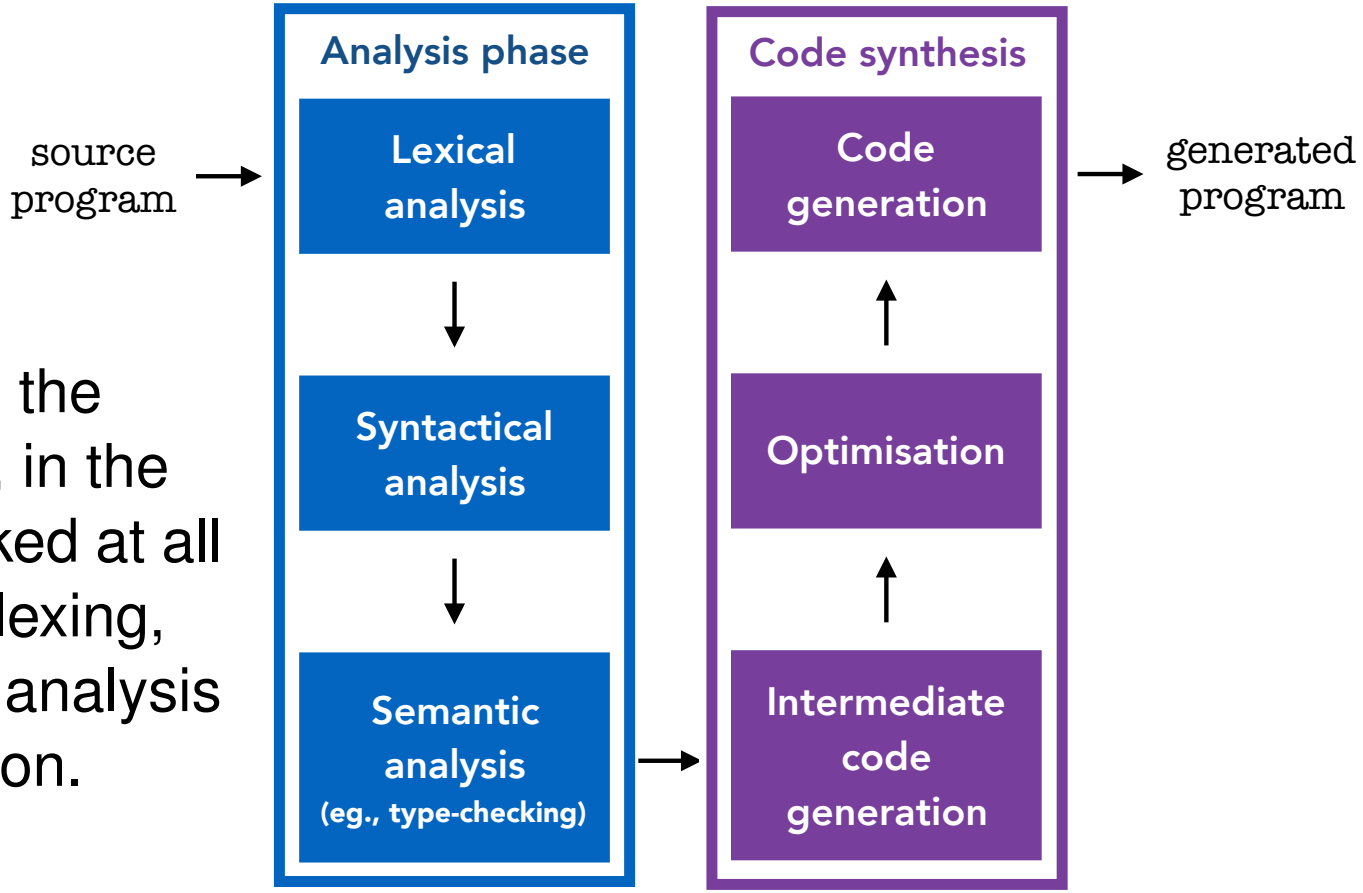

source program
Analysis phase
Lexical analysis
Syntactical analysis
Semantic analysis (eg., type-checking)
Code synthesis
Code generation
Optimisation
Intermediate code generation
generated program

Where we are

We have ‘finished’ the compilers module, in the sense that we looked at all compiler phases: lexing, parsing, semantic analysis and code generation.



Where we are



We have ‘finished’ the compilers module, in the sense that we looked at all compiler phases: lexing, parsing, semantic analysis and code generation.

We now take one final survey of how to manage memory in programs (aside from how to manage the heap), with an eye of trying to make memory access as fast as possible.

Introduction

We looked at code generation, but mostly considered a setting with simplifications to highlight the key ideas.

Introduction

We looked at code generation, but mostly considered a setting with simplifications to highlight the key ideas.

- We mostly considered very simple source languages: no procedures, no objects etc.

Introduction

We looked at code generation, but mostly considered a setting with simplifications to highlight the key ideas.

- We mostly considered very simple source languages: no procedures, no objects etc.
  - Although we briefly described how methods and other object-oriented features can be realised.

Introduction

We looked at code generation, but mostly considered a setting with simplifications to highlight the key ideas.

- We mostly considered very simple source languages: no procedures, no objects etc.
  - Although we briefly described how methods and other object-oriented features can be realised.
- Target language was simplistic (e.g. stack machine, or register machine with unbounded registers).

Introduction

We looked at code generation, but mostly considered a setting with simplifications to highlight the key ideas.

- We mostly considered very simple source languages: no procedures, no objects etc.
  - Although we briefly described how methods and other object-oriented features can be realised.
- Target language was simplistic (e.g. stack machine, or register machine with unbounded registers).
- Translation produces slow executables (in the case of stack- and accumulator machines), also we looked at one simple way to improve register allocation.

Key issues in translation to a realistic processor: speed

The code generators we've discussed work fine, even on real processors. The problem is speed of executables.

Key issues in translation to a realistic processor: speed

The code generators we've discussed work fine, even on real processors. The problem is speed of executables.

Naive code generation schemes make many memory reference, that can be avoided with better generated code.

Key issues in translation to a realistic processor: speed

The code generators we've discussed work fine, even on real processors. The problem is speed of executables.

Naive code generation schemes make many memory reference, that can be avoided with better generated code.

Why are memory references a problem? Because they are **slow** in comparison with register access (approx. two orders of magnitude slower).

Key issues in translation to a realistic processor: speed

The code generators we've discussed work fine, even on real processors. The problem is speed of executables.

Naive code generation schemes make many memory reference, that can be avoided with better generated code.

Why are memory references a problem? Because they are **slow** in comparison with register access (approx. two orders of magnitude slower).

But registers are **scarce** in real processors (core RISC-V has 32 integer registers). Putting a lot of effort into making best possible of registers pays off handsomely in executable speed.

Key issues in translation to a realistic processor: memory, energy

More generally a lot of effort goes into optimising the generated executable, not just for speed, but sometimes for memory, and, increasingly, for energy efficiency (important for battery powered devices, and for large-scale data centers).

Key issues in translation to a realistic processor: memory, energy

More generally a lot of effort goes into optimising the generated executable, not just for speed, but sometimes for memory, and, increasingly, for energy efficiency (important for battery powered devices, and for large-scale data centers).

For example on some architectures caching a computed value in memory and reusing it requires more energy than recomputing the value multiple times in registers.

Key issues in translation to a realistic processor: processor evolution

Processors evolve quickly. Extreme example: ARM processors in our phones.

Key issues in translation to a realistic processor: processor evolution

Processors evolve quickly. Extreme example: ARM processors in our phones.

In modern compilers up to 90% of compiler LOCs is optimisation. It would be annoying if we had to rewrite this every time a processor evolves a bit.

Key issues in translation to a realistic processor: processor evolution

Processors evolve quickly. Extreme example: ARM processors in our phones.

In modern compilers up to 90% of compiler LOCs is optimisation. It would be annoying if we had to rewrite this every time a processor evolves a bit.

Fortunately, most machine languages are fairly similar.

Key issues in translation to a realistic processor: processor evolution

Processors evolve quickly. Extreme example: ARM processors in our phones.

In modern compilers up to 90% of compiler LOCs is optimisation. It would be annoying if we had to rewrite this every time a processor evolves a bit.

Fortunately, most machine languages are fairly similar.

So we compile first to an intermediate representation (IR) which is rather close to generic machine code, but hides some detail.

- Most optimisation is done on this IR.
- Target machine Code is easy to generate from the IR.

Key issues in translation to a realistic processor: processor evolution

Processors evolve quickly. Extreme example: ARM processors in our phones.

In modern compilers up to 90% of compiler LOCs is optimisation. It would be annoying if we had to rewrite this every time a processor evolves a bit.

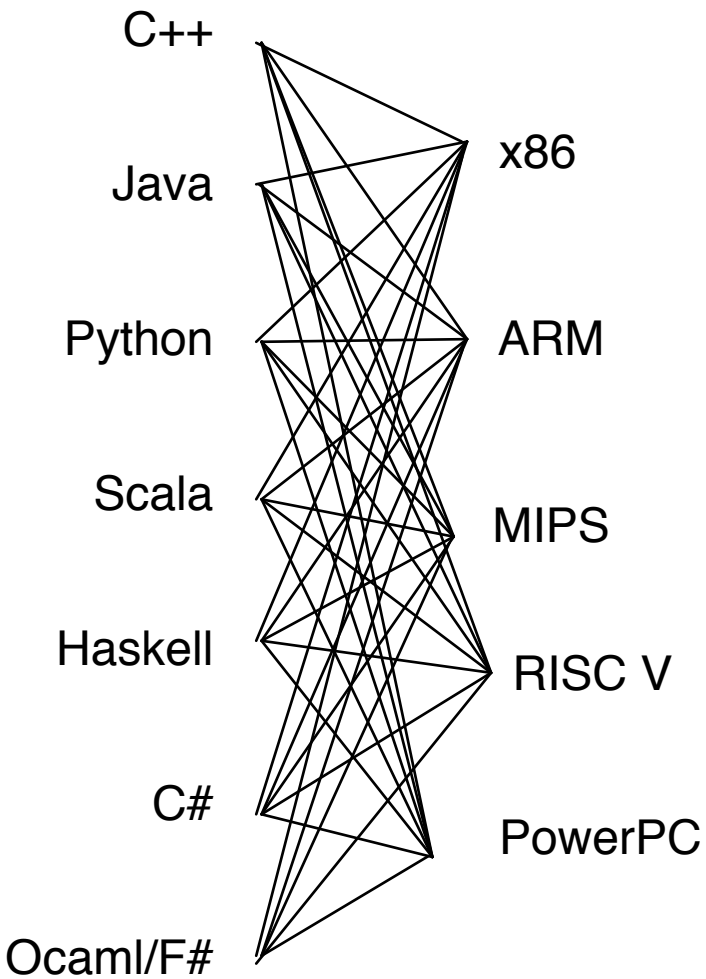
Fortunately, most machine languages are fairly similar.

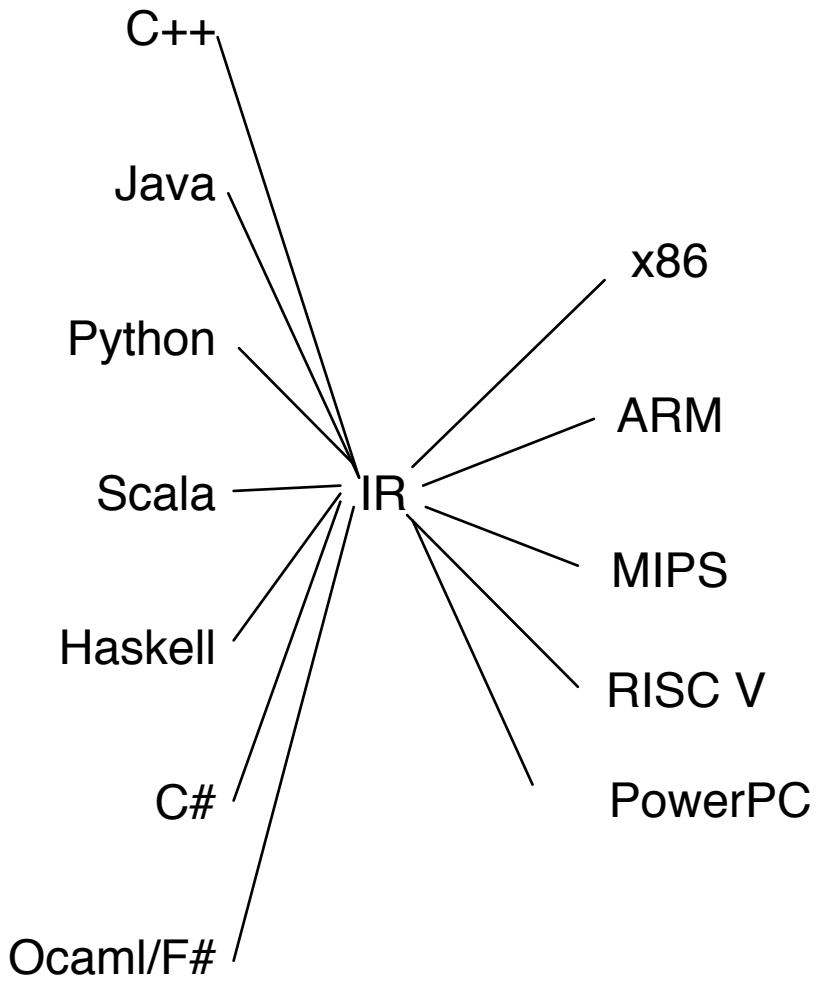
So we compile first to an intermediate representation (IR) which is rather close to generic machine code, but hides some detail.

- Most optimisation is done on this IR.
- Target machine Code is easy to generate from the IR.

A picture says more than 1000 words.

Key issues in translation to a realistic processor: processor evolution





Intermediate representation

A good intermediate representation should have the following properties.

- IR must be easy to generate for front-end.

Intermediate representation

A good intermediate representation should have the following properties.

- IR must be easy to generate for front-end.
- IR must be easy to convert to real machine code for all desired target machines.

Intermediate representation

A good intermediate representation should have the following properties.

- IR must be easy to generate for front-end.
- IR must be easy to convert to real machine code for all desired target machines.
- Each construct in the IR must have a clear and simple meaning, so that optimisation can easily be specified and implemented.

Intermediate representation

A good intermediate representation should have the following properties.

- IR must be easy to generate for front-end.
- IR must be easy to convert to real machine code for all desired target machines.
- Each construct in the IR must have a clear and simple meaning, so that optimisation can easily be specified and implemented. (Optimisations in real compilers are complicated and regularly introduce errors.)

Intermediate representation

A good intermediate representation should have the following properties.

- IR must be easy to generate for front-end.
- IR must be easy to convert to real machine code for all desired target machines.
- Each construct in the IR must have a clear and simple meaning, so that optimisation can easily be specified and implemented. (Optimisations in real compilers are complicated and regularly introduce errors.)

Real compilers often even use multiple IRs.

Intermediate representation: LLVM (not relevant for exams)

In 2022, the most well-known and widely used IR is probably LLVM (used to stand for “Low-Level Virtual Machine” although the project is now way bigger than virtual machines). It was developed by a student (Chris Lattner) for his final year dissertation. C. Lattner went to Apple where LLVM is now used in all Apple dev tools (OS X, iOS), did `clang`, Swift, led Autopilot Software at Tesla, led the TensorFlow Infrastructure team at Google... He’s now leading the Engineering and Product team at SiFive.

Intermediate representation: LLVM (not relevant for exams)

In 2022, the most well-known and widely used IR is probably LLVM (used to stand for “Low-Level Virtual Machine” although the project is now way bigger than virtual machines). It was developed by a student (Chris Lattner) for his final year dissertation. C. Lattner went to Apple where LLVM is now used in all Apple dev tools (OS X, iOS), did `clang`, Swift, led Autopilot Software at Tesla, led the TensorFlow Infrastructure team at Google... He’s now leading the Engineering and Product team at SiFive.

LLVM is free and open source, and has evolved into large tool suite for building compilers. I recommend that you take a look: `http://llvm.org`

Intermediate representation: LLVM (not relevant for exams)

In 2022, the most well-known and widely used IR is probably LLVM (used to stand for “Low-Level Virtual Machine” although the project is now way bigger than virtual machines). It was developed by a student (Chris Lattner) for his final year dissertation. C. Lattner went to Apple where LLVM is now used in all Apple dev tools (OS X, iOS), did `clang`, Swift, led Autopilot Software at Tesla, led the TensorFlow Infrastructure team at Google... He’s now leading the Engineering and Product team at SiFive.

LLVM is free and open source, and has evolved into large tool suite for building compilers. I recommend that you take a look: `http://llvm.org`

If you want / need to build an optimising compiler, my recommendation is to use LLVM as backend.

Intermediate representation: WebAssembly (not relevant for exams)

Last few years saw the rise of compile-to-Javascript, e.g. TypeScript, Reason.ml and Scala.js.

Javascript is not a good language to compile to.

In the Summer 2017, all major browser vendors agreed to support WebAssembly as an platform independent IR.

Intermediate representation: WebAssembly (not relevant for exams)

Last few years saw the rise of compile-to-Javascript, e.g. TypeScript, Reason.ml and Scala.js.

Javascript is not a good language to compile to.

In the Summer 2017, all major browser vendors agreed to support WebAssembly as an platform independent IR.

WebAssembly's most interesting features:

- Typed
- Secure (e.g. no buffer overflow possible)

Intermediate representation: WebAssembly (not relevant for exams)

Last few years saw the rise of compile-to-Javascript, e.g. TypeScript, Reason.ml and Scala.js.

Javascript is not a good language to compile to.

In the Summer 2017, all major browser vendors agreed to support WebAssembly as an platform independent IR.

WebAssembly's most interesting features:

- Typed
- Secure (e.g. no buffer overflow possible)

Will WebAssembly succeed? Time will tell ... but things are looking good.

Check out: `http://webassembly.org`

Intermediate representation

Often IRs (including LLVM) are quite simple and look very similar to, e.g., register machine code with unlimited registers.

Intermediate representation

Often IRs (including LLVM) are quite simple and look very similar to, e.g., register machine code with unlimited registers.

Such IR is often called **three address code** because most instructions look something like this:

```
r1 := r2 op r3
```

Intermediate representation

Often IRs (including LLVM) are quite simple and look very similar to, e.g., register machine code with unlimited registers.

Such IR is often called **three address code** because most instructions look something like this:

```
r1 := r2 op r3
```

Here `r1`, `r2`, `r3` are registers and `op` is an operation like addition, comparison ... It is clear that this can easily be mapped to real assembly instructions.

Intermediate representation: SSA (not relevant for exams)

Most modern compilers use a variant of three address code called **SSA** form, short for **static single assignment** form.

- Each variable is assigned exactly once (what about loops?)
- Each varianble must be defined (= assigned to) before use

This format makes compiler optimisations easier and faster.

Intermediate representation: SSA (not relevant for exams)

Most modern compilers use a variant of three address code called **SSA** form, short for **static single assignment** form.

- Each variable is assigned exactly once (what about loops?)
- Each varianble must be defined (= assigned to) before use

This format makes compiler optimisations easier and faster.

Surprisingly close to functional programming.

Intermediate representation: PTX (not relevant for exams)

PTX (= Parallel Thread Execution) is an intermediate language used for compiling Nvidia's CUDA to Nvidia's GPUs.

It's a register machine architecture with unlimited registers.

Run-time organisation of memory

What happens when a program is invoked?

Run-time organisation of memory

What happens when a program is invoked?

- We ask the OS to allocate space for the program.

Run-time organisation of memory

What happens when a program is invoked?

- We ask the OS to allocate space for the program.
- The OS loads the code of the program into the allocated space. Maybe maps symbolic names to (virtual) memory addresses. This is called (dynamic) linking.

Run-time organisation of memory

What happens when a program is invoked?

- We ask the OS to allocate space for the program.
- The OS loads the code of the program into the allocated space. Maybe maps symbolic names to (virtual) memory addresses. This is called (dynamic) linking.
- The OS jumps to the entry point of the program, i.e. the (translation of the) program's entry point (e.g.: in Java the `main` method).

Saving caller data

```
def f (...) {
    x = x*y
    g( x, y, z )
    x = z+y+z }
```

If `f` calls `g`, then `g` typically uses registers. But these registers are typically also used by the activation of `f` that is not yet finished.

Saving caller data

```
def f (...) {
    x = x*y
    g( x, y, z )
    x = z+y+z }
```

If `f` calls `g`, then `g` typically uses registers. But these registers are typically also used by the activation of `f` that is not yet finished.

It would be bad if the activation of `g` overwrote `f`'s registers.

Saving caller data

```
def f (...) {
    x = x*y
    g( x, y, z )
    x = z+y+z }
```

If `f` calls `g`, then `g` typically uses registers. But these registers are typically also used by the activation of `f` that is not yet finished.

It would be bad if the activation of `g` overwrote `f`'s registers.

What happens is that the call to `g` **saves** `f`'s registers before executing `g`, and when the execution of `g` is finished, these saved registers are **restored**. The saved registers are also stored in `g`'s AR.

Saving caller data

```
def f (...) {
    x = x*y
    g( x, y, z )
    x = z+y+z }
```

If `f` calls `g`, then `g` typically uses registers. But these registers are typically also used by the activation of `f` that is not yet finished.

It would be bad if the activation of `g` overwrote `f`'s registers.

What happens is that the call to `g` **saves** `f`'s registers before executing `g`, and when the execution of `g` is finished, these saved registers are **restored**. The saved registers are also stored in `g`'s AR.

The compiler must generate the code to save and restore caller registers. More about this later.

Note that the AR layout just sketched, and division of responsibility between the caller and callee are contingent (could have been done otherwise).

What conventions we use depends on many factors, such as:

- Compatibility with existing software.
- Availability of CPU support for some operations.
- Speed or memory usage.
- Code generation simplicity.

Non-integer procedure arguments

What we have not covered is procedures taking non-integer arguments.

Non-integer procedure arguments

What we have not covered is procedures taking non-integer arguments.

The only difference from a code generation perspective between integer types and other types as procedure arguments is the **size** of the data. But that size is known at compile-time (at least for languages that are statically typed).

Non-integer procedure arguments

What we have not covered is procedures taking non-integer arguments.

The only difference from a code generation perspective between integer types and other types as procedure arguments is the **size** of the data. But that size is known at compile-time (at least for languages that are statically typed).

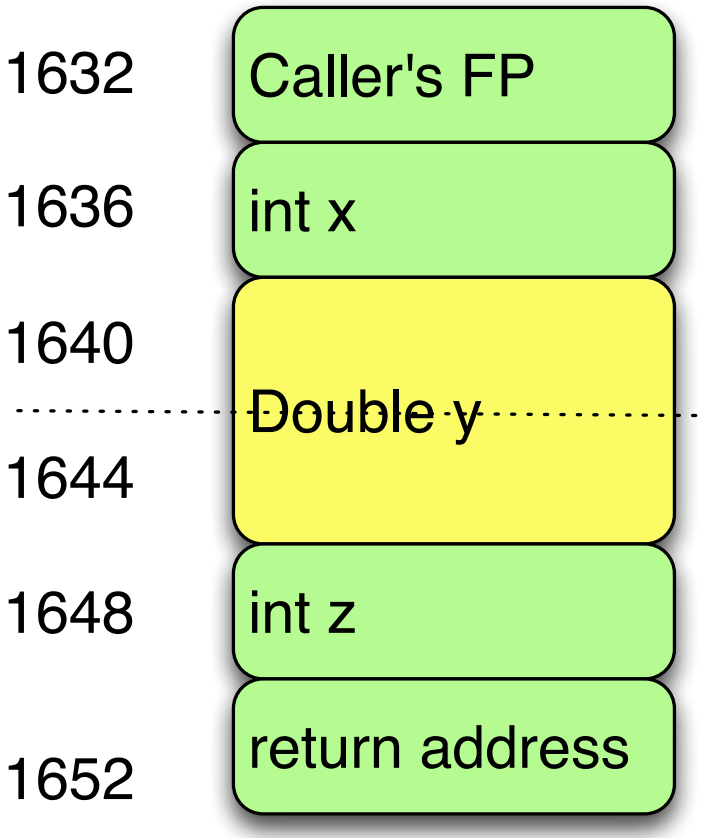
For example the type `double` is often 64 bits. So we reserve 8 bytes for arguments of that type in the procedure's AR layout. We may have to use two calls to `lw` and `sw` to load and store such arguments, but otherwise code generation is unchanged.

Non-integer procedure arguments

Consider a procedure with the following signature:

```
int f ( int x,
        double y,
        int z ) = { ... }
```

Assuming that `double` is stored as 64 bits, then the AR would look like on the right.

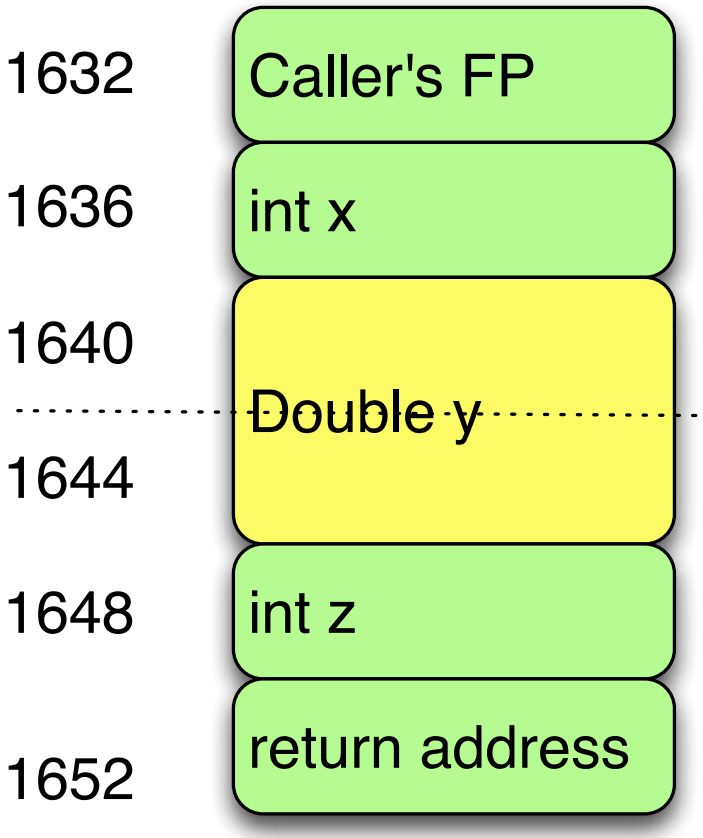


Non-integer procedure arguments

Consider a procedure with the following signature:

```
int f ( int x,
        double y,
        int z ) = { ... }
```

Assuming that `double` is stored as 64 bits, then the AR would look like on the right.



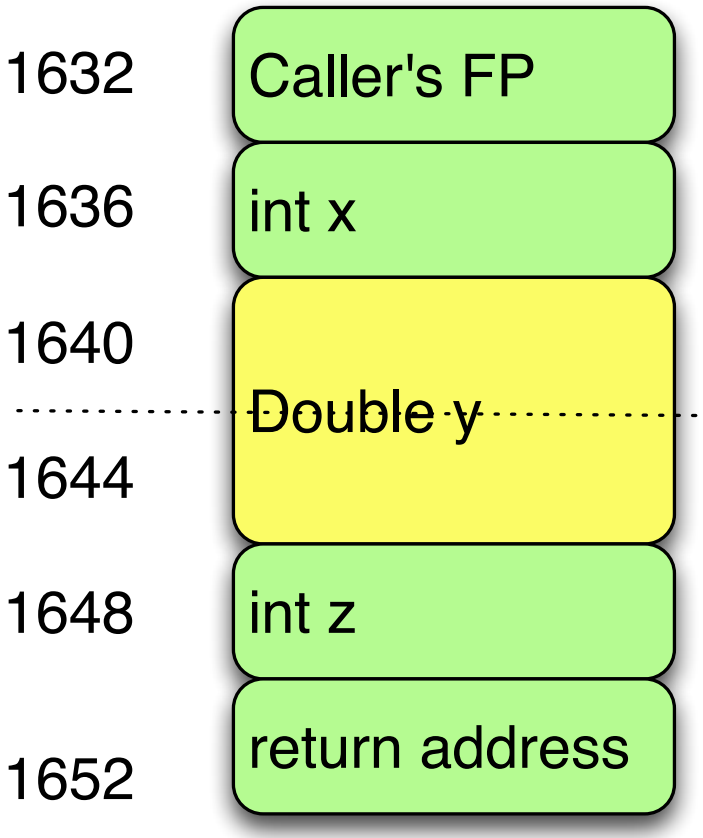
How does the code generator know the size of the variables?

Non-integer procedure arguments

Consider a procedure with the following signature:

```
int f ( int x,
        double y,
        int z ) = { ... }
```

Assuming that `double` is stored as 64 bits, then the AR would look like on the right.



How does the code generator know the size of the variables?

Use the information stored in the symbol table, which was created by the type checker and passed to the code generator.

Non-integer procedure arguments

If we use a simplistic ‘accumulator machine’ approach, cannot do the same with the return value, e.g.

```
double f ( int x, double y, int z ) = ...
```

Non-integer procedure arguments

If we use a simplistic ‘accumulator machine’ approach, cannot do the same with the return value, e.g.

```
double f ( int x, double y, int z ) = ...
```

This is because the accumulator holds the return value of procedure calls, and the accumulator is fixed at 32 bits.

Non-integer procedure arguments

If we use a simplistic ‘accumulator machine’ approach, cannot do the same with the return value, e.g.

```
double f ( int x, double y, int z ) = ...
```

This is because the accumulator holds the return value of procedure calls, and the accumulator is fixed at 32 bits.

In this case we’d have to move to an approach that holds the return value also in the AR (either for all arguments, or only for arguments that don’t fit in a register – we know at compile time which is which).

Interesting observations (1)

Stack allocated memory is much **faster** than heap allocation, because (1) acquiring stack memory is just a constant-time push operation, and (2) the whole AR can be 'deleted' (= popped off the stack) in a single, constant-time operation.

Automatic heap de-allocation (garbage collection) is especially expensive. This is why low-level language (C, C++, Rust) don't have garbage collection (by default).

ARs in real compilers

Production compilers attempt to hold as much of the AR in registers as possible, especially the procedure arguments and result. In other words: in this case the AR is stored partly on the stack and partly in registers.

ARs in real compilers

Production compilers attempt to hold as much of the AR in registers as possible, especially the procedure arguments and result. In other words: in this case the AR is stored partly on the stack and partly in registers.

Reason: speed!

ARs in real compilers

Production compilers attempt to hold as much of the AR in registers as possible, especially the procedure arguments and result. In other words: in this case the AR is stored partly on the stack and partly in registers.

Reason: speed!

This can make procedure invocations complicated to compile.

Available memory

	Capacity	Latency	Cost
Register	1000s of bytes	1 cycle	£££££
SRAM	1s of MBytes	several cycles	????
DRAM	10s GBytes	20 - 100 cycles	??
Flash	100s of GBytes		?
Hard disk	10 TByte	0.5 - 5 M cycles	cheap
Ideal	1000s GBytes	1 cycle	cheap

- RAM = Random Access Memory
- SRAM = static RAM, fast but uses 6 transistors per bit.
- DRAM = dynamic RAM, slow but uses 1 transistor per bit.
- Flash = non-volatile, slow, loses capacity over time.
- Latency is the time between issuing the read/write access and the completion of this access.

Available memory

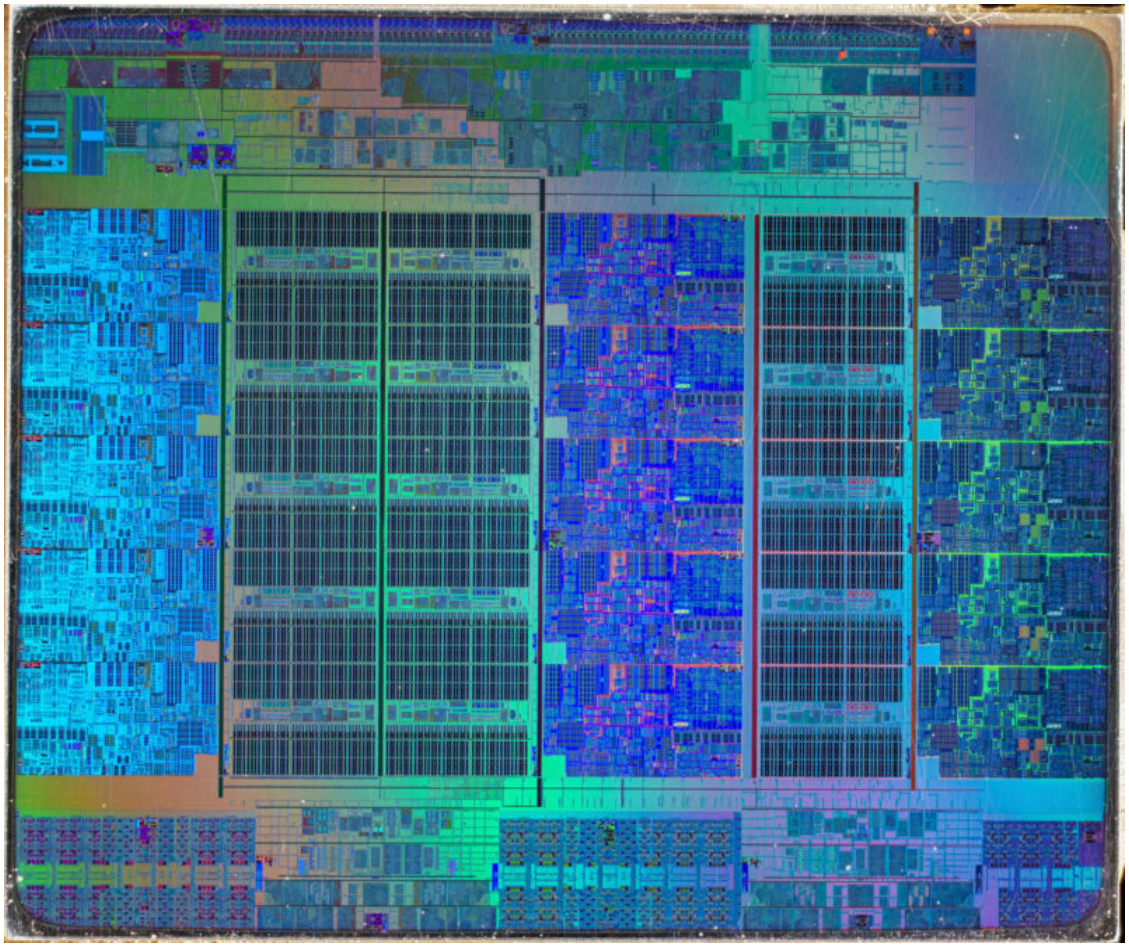
	Capacity	Latency	Cost
Register	1000s of bytes	1 cycle	£££££
SRAM	1s of MBytes	several cycles	????
DRAM	10s GBytes	20 - 100 cycles	??
Flash	100s of GBytes		?
Hard disk	10 TByte	0.5 - 5 M cycles	cheap
Ideal	1000s GBytes	1 cycle	cheap

- RAM = Random Access Memory
- SRAM = static RAM, fast but uses 6 transistors per bit.
- DRAM = dynamic RAM, slow but uses 1 transistor per bit.
- Flash = non-volatile, slow, loses capacity over time.
- Latency is the time between issuing the read/write access and the completion of this access.

It seems memory is either small, fast and expensive, or cheap, big and slow.

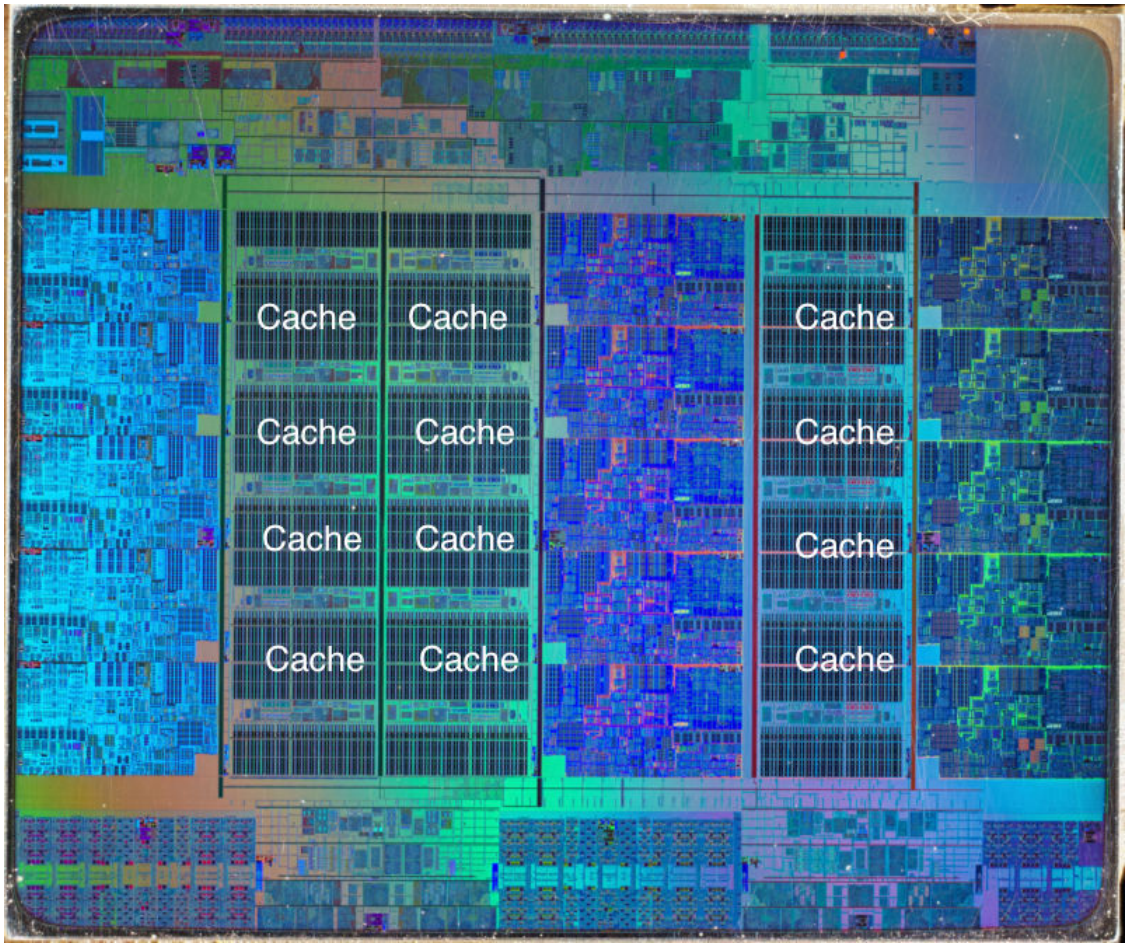
Caches in modern CPUs

Let's look at a modern CPU. Here is a November 2018 Intel Ivy Bridge Xeon CPU. Much of the silicon is for the cache, and cache controllers.

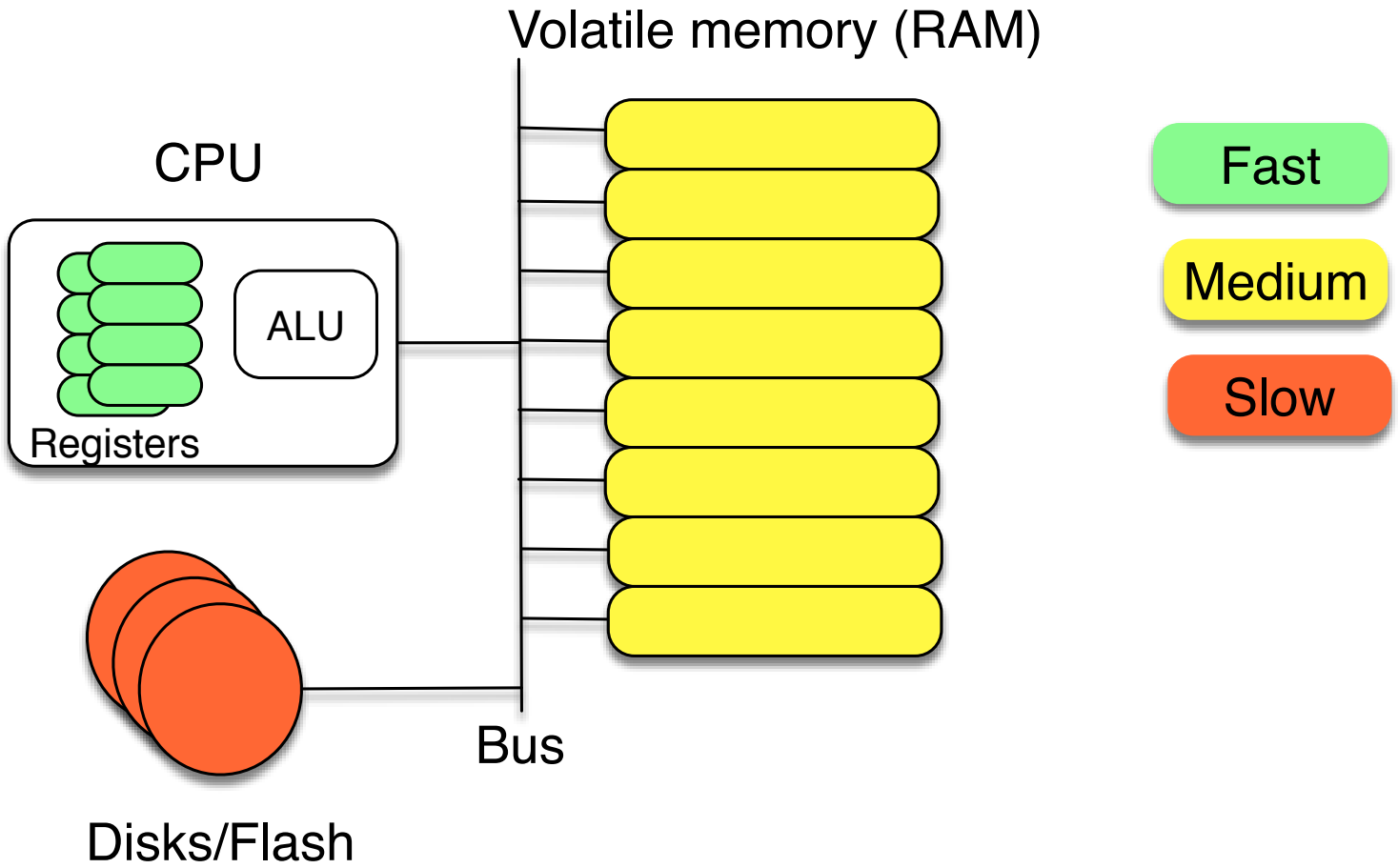


Caches in modern CPUs

Why is much of the chip area dedicated to caches?



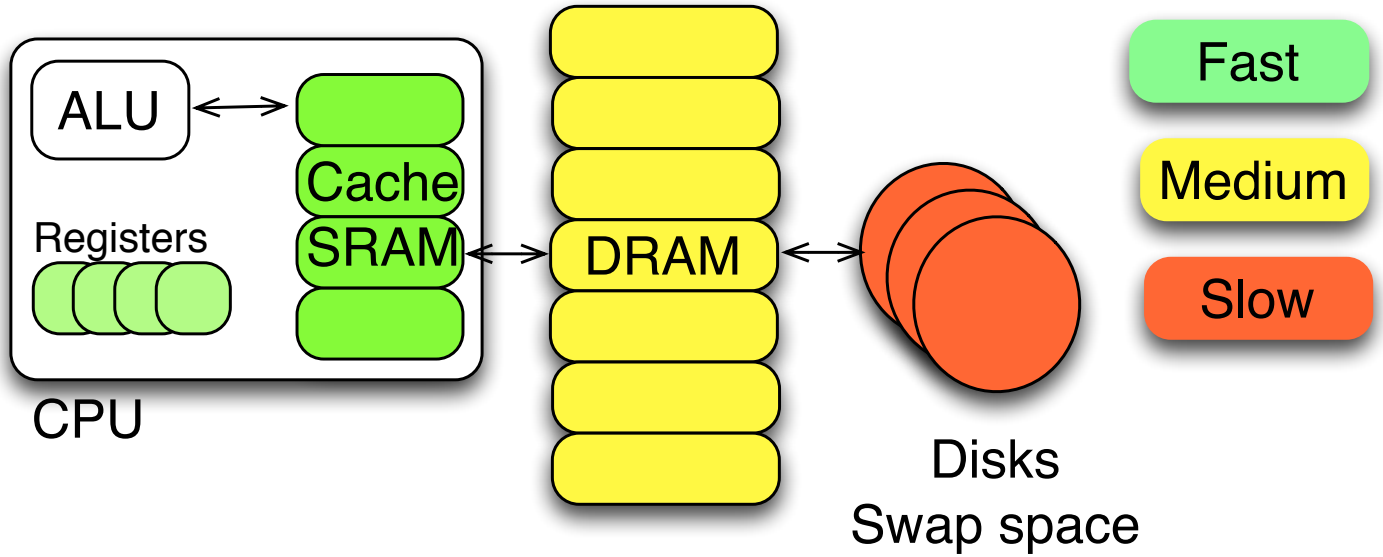
Simplified computer layout



The faster the memory, the faster the computer.

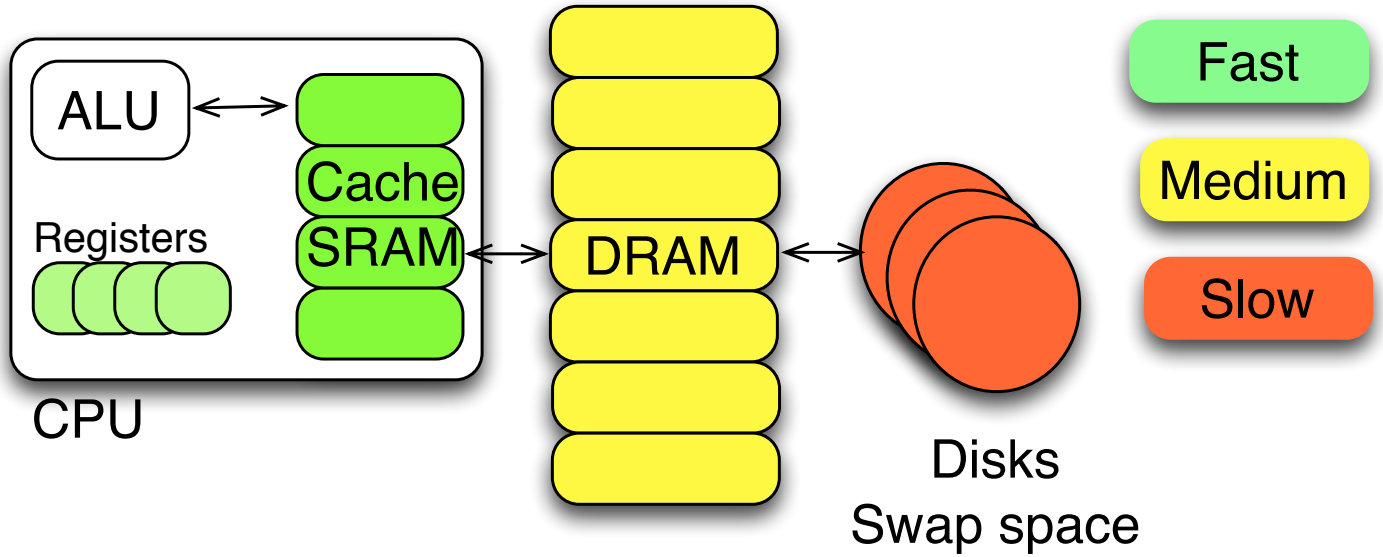
Cache

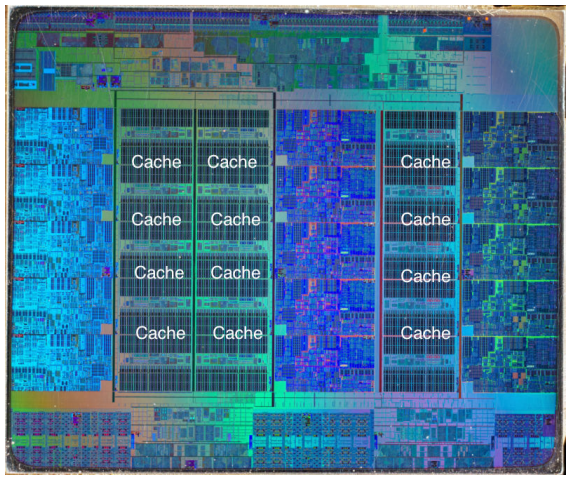
A crucial core element of modern CPUs.



Cache

A crucial core element of modern CPUs.





Tricks for making the computer run faster

Use a **hierarchy** of memory technology, in particular,

- Keep the most often-used data in a small, fast memory.
- Use slow main memory only rarely.

Tricks for making the computer run faster

Use a **hierarchy** of memory technology, in particular,

- Keep the most often-used data in a small, fast memory.
- Use slow main memory only rarely.

But how to achieve these goals?

Tricks for making the computer run faster

Use a **hierarchy** of memory technology, in particular,

- Keep the most often-used data in a small, fast memory.
- Use slow main memory only rarely.

But how to achieve these goals? Does that mean programmers have to worry about this?

Tricks for making the computer run faster

Use a **hierarchy** of memory technology, in particular,

- Keep the most often-used data in a small, fast memory.
- Use slow main memory only rarely.

But how to achieve these goals? Does that mean programmers have to worry about this? Let the CPU worry about this, using a mechanism called **cache**.

Tricks for making the computer run faster

Use a **hierarchy** of memory technology, in particular,

- Keep the most often-used data in a small, fast memory.
- Use slow main memory only rarely.

But how to achieve these goals? Does that mean programmers have to worry about this? Let the CPU worry about this, using a mechanism called **cache**.

Why on earth would this work?

Tricks for making the computer run faster

Use a **hierarchy** of memory technology, in particular,

- Keep the most often-used data in a small, fast memory.
- Use slow main memory only rarely.

But how to achieve these goals? Does that mean programmers have to worry about this? Let the CPU worry about this, using a mechanism called **cache**.

Why on earth would this work?

Exploit **locality!**

Locality

In practice, most programs, most of the time, do the following:

Locality

In practice, most programs, most of the time, do the following:

> *If the program accesses memory location* `loc` *then the most of the next few memory accesses are very likely at addresses close to* `loc`*.*

Locality

In practice, most programs, most of the time, do the following:

> *If the program accesses memory location* `loc` *then the most of the next few memory accesses are very likely at addresses close to* `loc`*.*

I.e. we use access to a memory location as a **heuristic prediction** for memory access in the (near) future. This is called **(data) locality**. Locality means that memory access often (but not necessarily always) follows this fairly **predictable** pattern in practice.

Modern CPUs exploit this predictability for speed with caches.

Locality

In practice, most programs, most of the time, do the following:

> *If the program accesses memory location* `loc` *then the most of the next few memory accesses are very likely at addresses close to* `loc`*.*

I.e. we use access to a memory location as a **heuristic prediction** for memory access in the (near) future. This is called **(data) locality**. Locality means that memory access often (but not necessarily always) follows this fairly **predictable** pattern in practice.

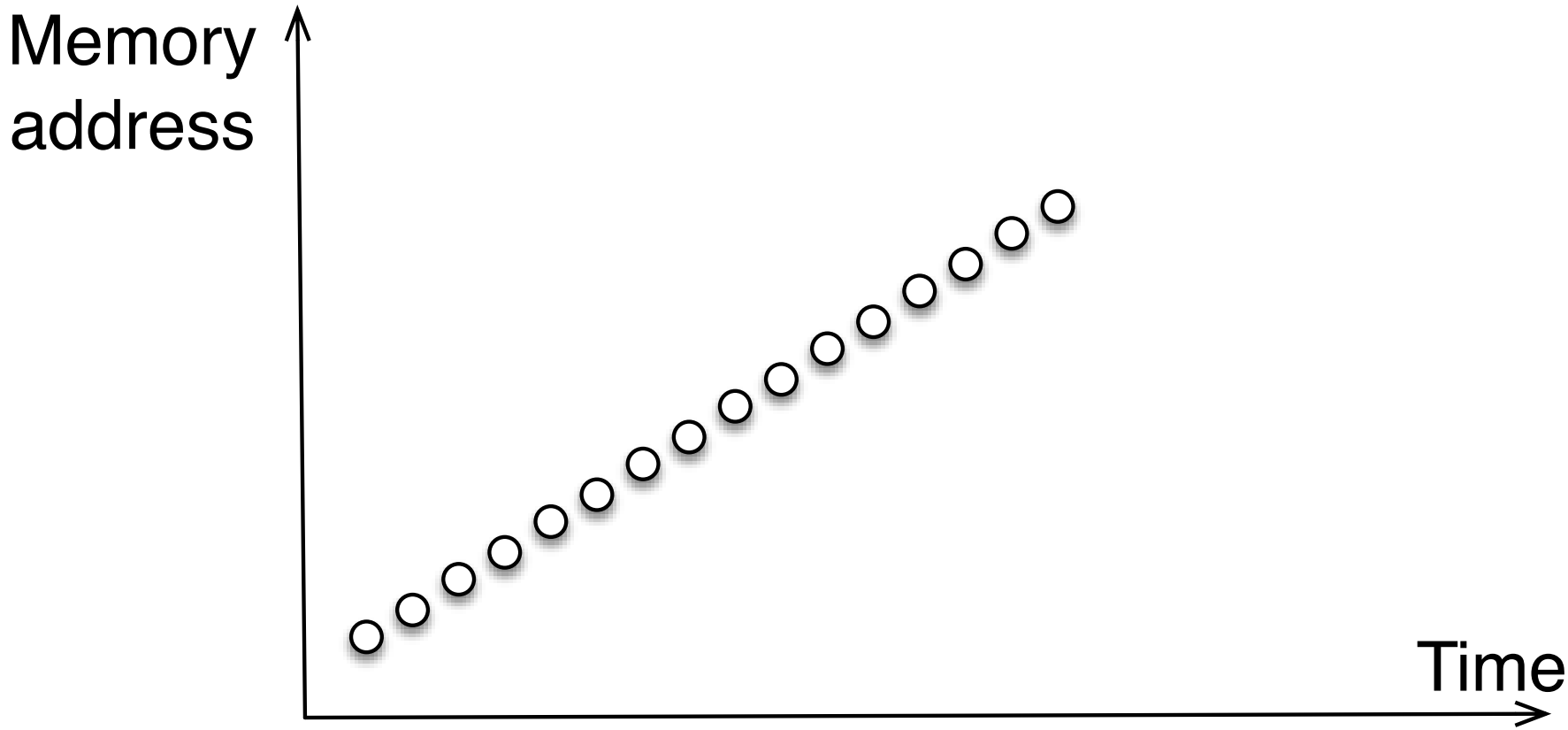
Modern CPUs exploit this predictability for speed with caches.

Note: it is possible to write programs that don't exhibit locality. They will typically run very slow.

Why would most programs exhibit locality?

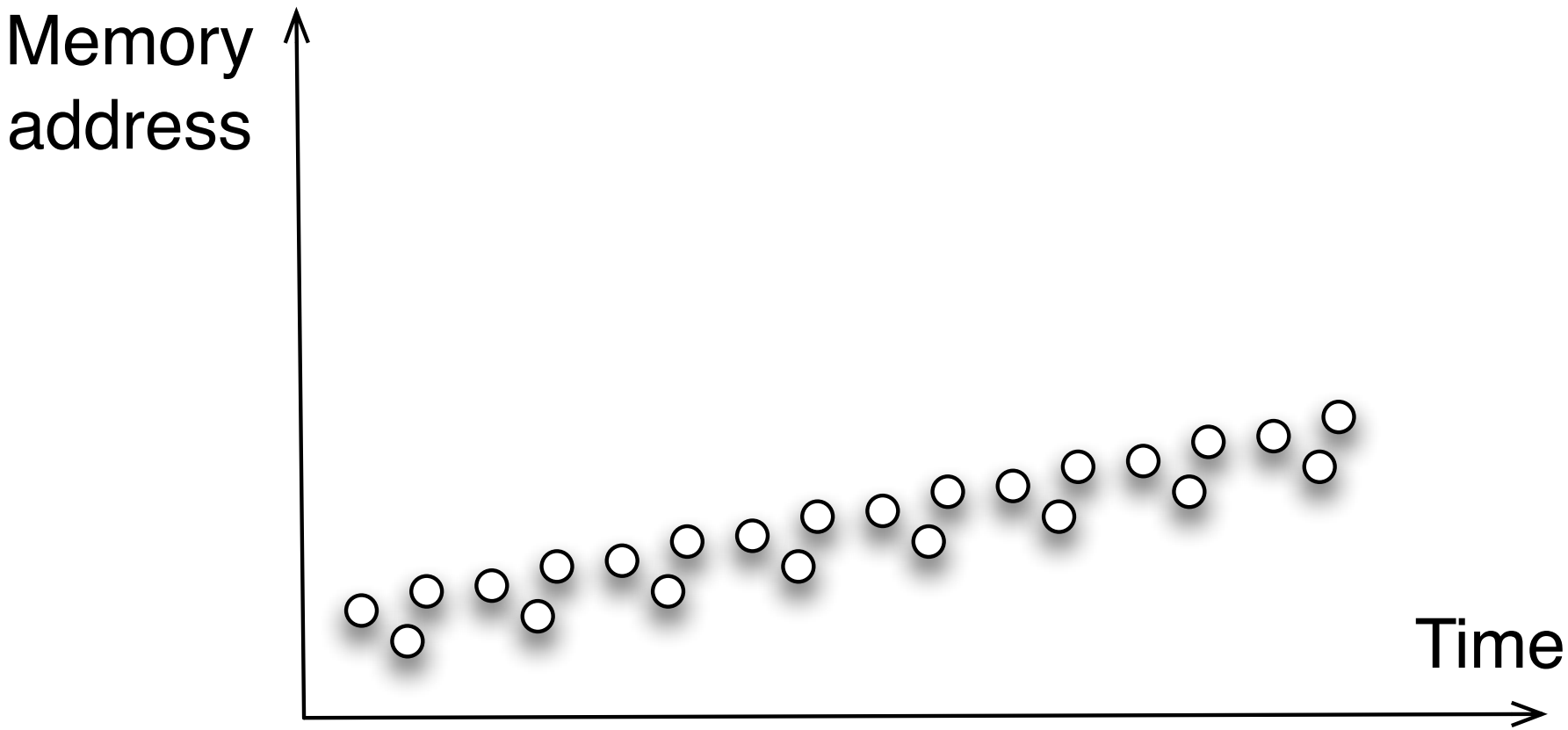
Data locality

```
int [] a = new int [1000000];
for ( int i = 0; i < 1000000; i++ ) {
    a [ i ] = i+1; }
```



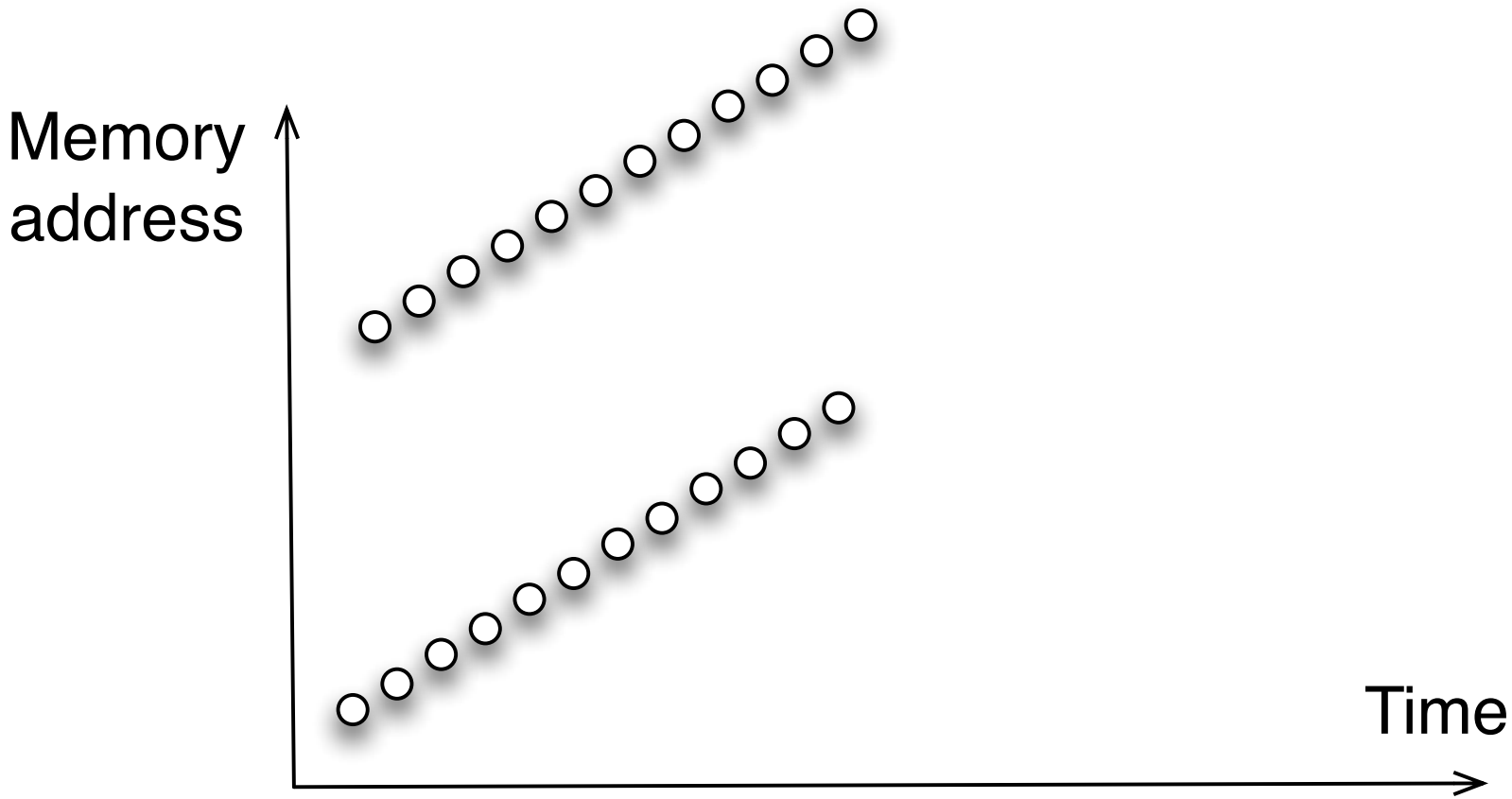
Data locality

```
int [] a = new int [1000000];
...
for ( int i = 2; i < 1000000; i++ ) {
   a [ i ] = a [i-1] * a [i-2]; }
```



Data locality

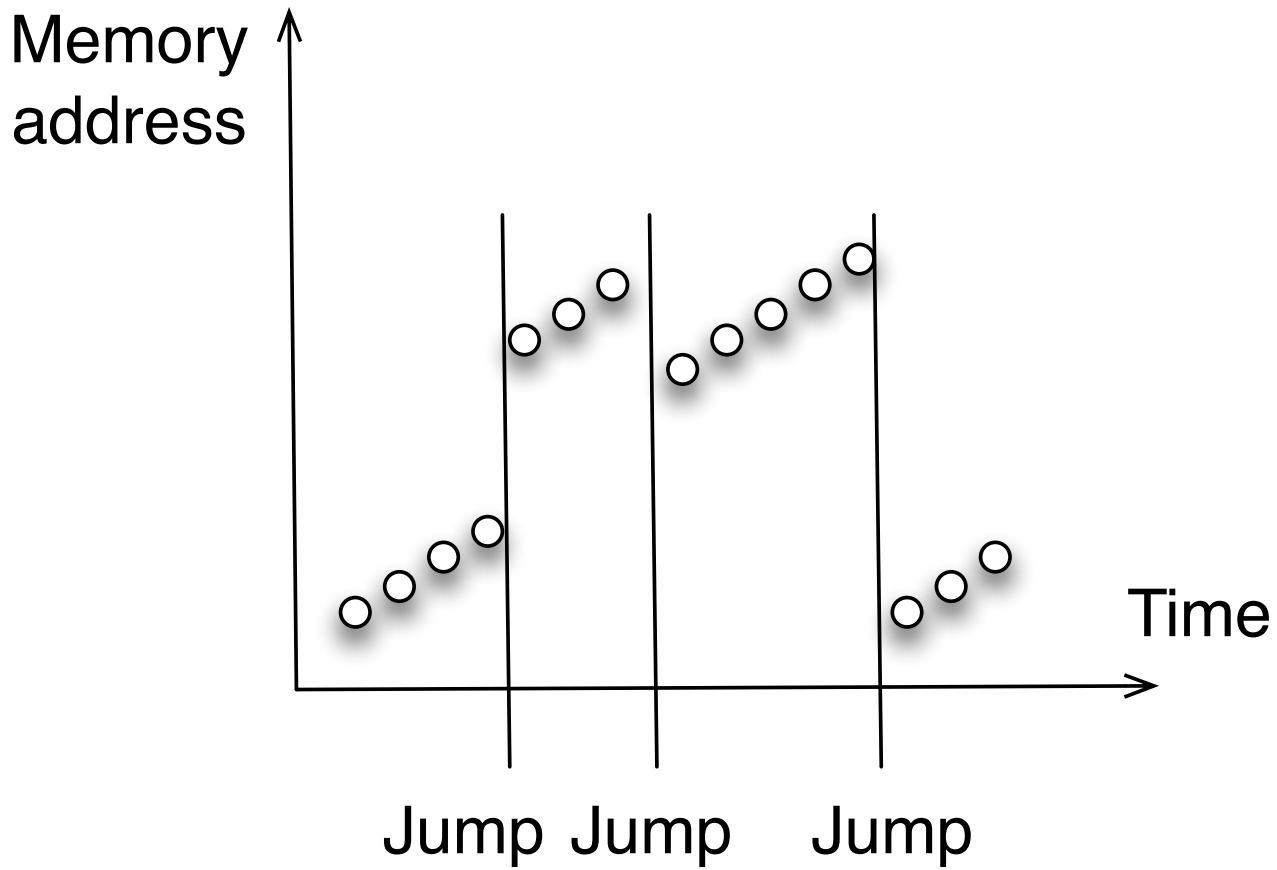
```
int [] a = new int [1000000];
int [] b = new int [1000000];
...
for ( int i = 0; i < 1000000; i++ ) {
   a [ i ] = b [ i ] + 1; }
```



Code locality

Program execution (reading via PC) is local too, with occasional jumps.

```
addiu  sp  sp
li  a0 1
lw  t1 4(sp)
sub  a0  t1  a
addiu  sp  sp
sw  a0 0( sp)
addiu  sp  sp
jal next
lw  t1 4(sp)
add  a0  t1  a
addiu  sp  sp
b exit2
```

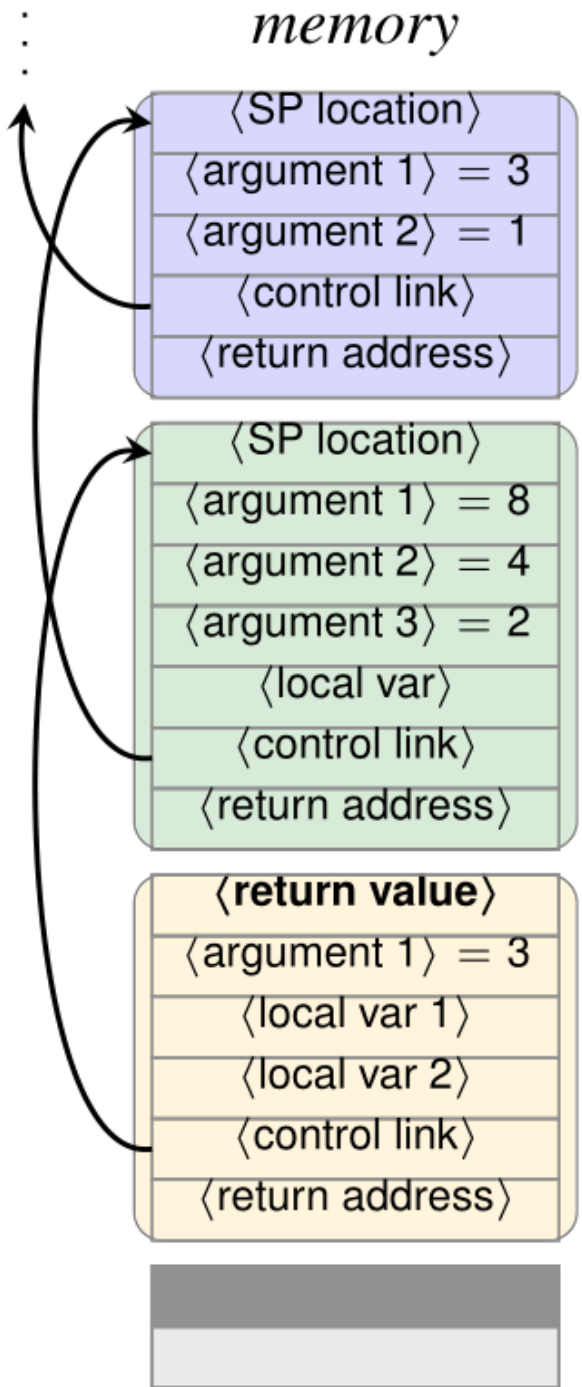


Data locality

Another cause for data locality is the stack and how we compile procedure invocations into activation records.

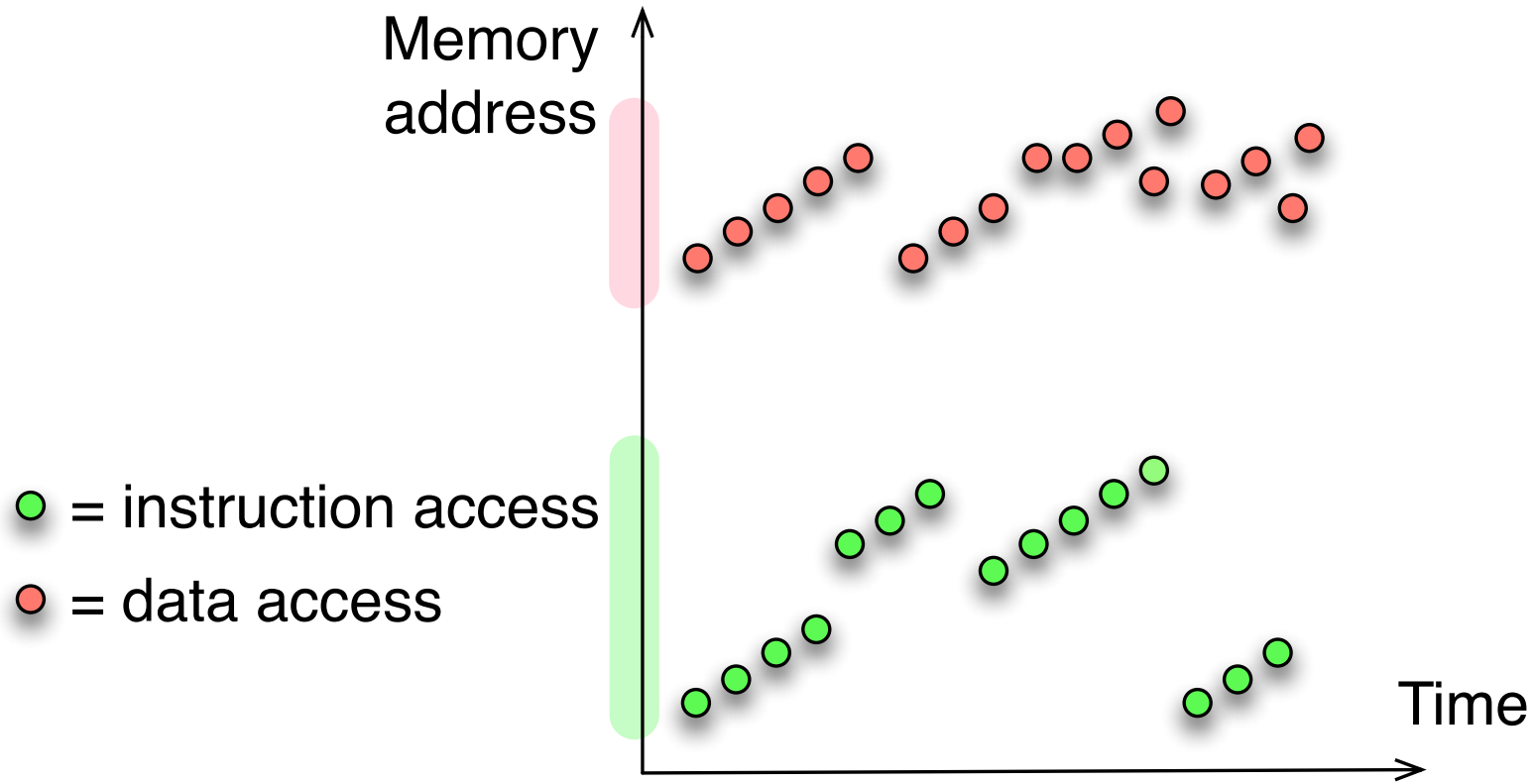
This is because within a procedure activation we typically spend a lot of time accessing the procedure arguments and local variables.

In addition, in recursive procedure invocations, related activation records, are nearby on the stack.



Locality

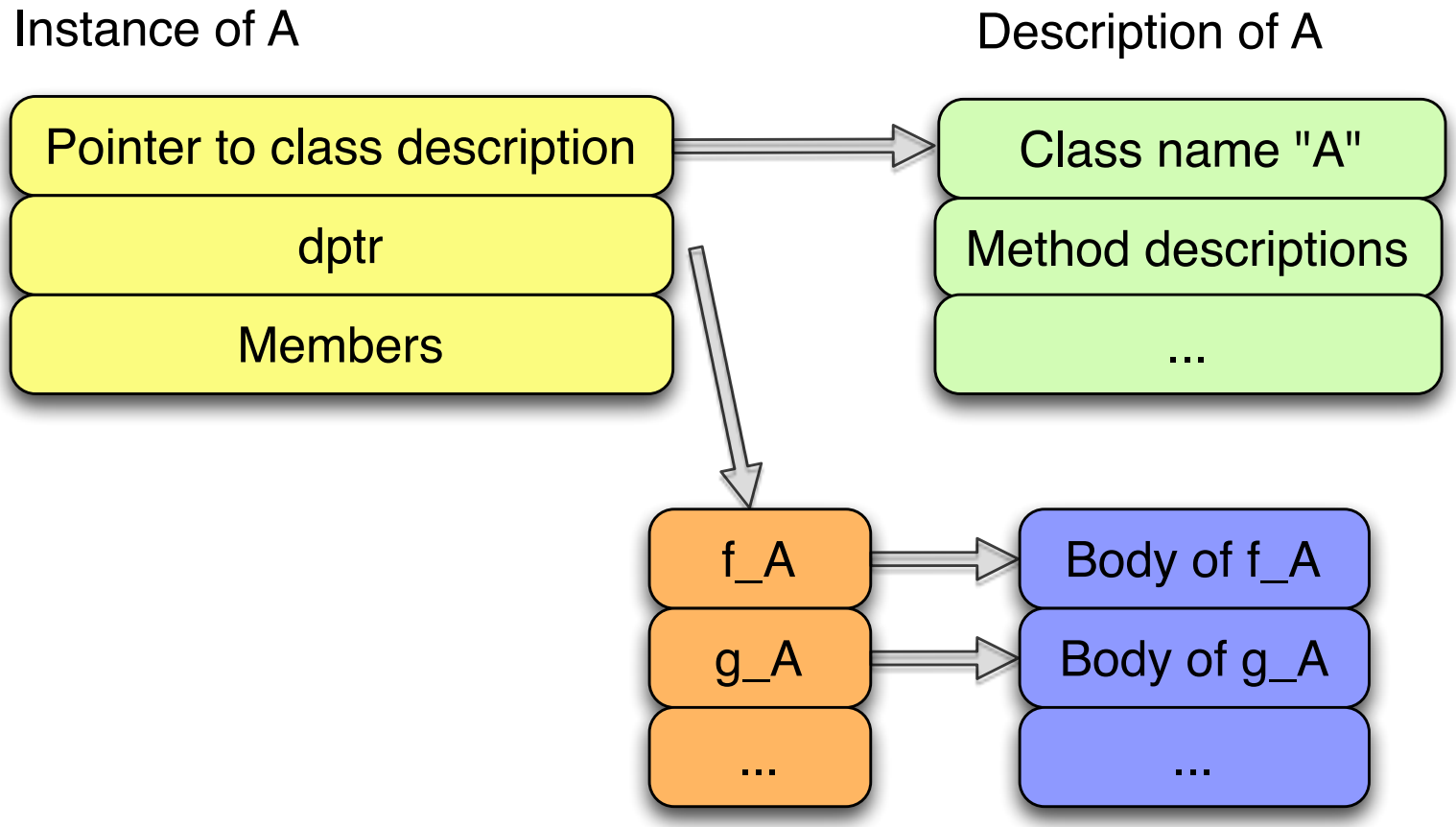
In practice we have data access and instruction access together, so the access patterns look more like this:



Still a lot of predictability in memory access patterns, but over (at least) two distinct regions of memory.

Data locality of OO programming

Accessing objects, especially method invocation often has **bad** locality because of pointer chasing. Object pointers can point anywhere inside the heap, losing locality.



To ameliorate this shortcoming, JIT compilers have been developed..

How to exploit locality and the memory hierarchy?

Two approaches:

How to exploit locality and the memory hierarchy?

Two approaches:

- **Expose** the hierarchy (proposed by S. Cray): let programmers access registers, fast SRAM, slow DRAM and the disk 'by hand'. Tell them "Use the hierarchy cleverly". This is already out of fashion very long ago.
- **Hide** the memory hierarchy. Programming model: there is a **single** kind of memory, single address space (excluding registers). **Automatically** assigns locations to fast or slow memory, depending on usage patterns. This is what **caches** do in CPUs.

Cache

A CPU **cache** is used by the CPU of a computer to reduce the average time to access memory. The cache is a smaller, faster and more expensive memory inside the CPU which stores copies of the data from the **most frequently** used main memory locations for fast access. A cache entry is called **cache line**.

Cache

A CPU **cache** is used by the CPU of a computer to reduce the average time to access memory. The cache is a smaller, faster and more expensive memory inside the CPU which stores copies of the data from the **most frequently** used main memory locations for fast access. A cache entry is called **cache line**.

Cache

A CPU **cache** is used by the CPU of a computer to reduce the average time to access memory. The cache is a smaller, faster and more expensive memory inside the CPU which stores copies of the data from the **most frequently** used main memory locations for fast access. A cache entry is called **cache line**.

When the CPU reads from or writes to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory.

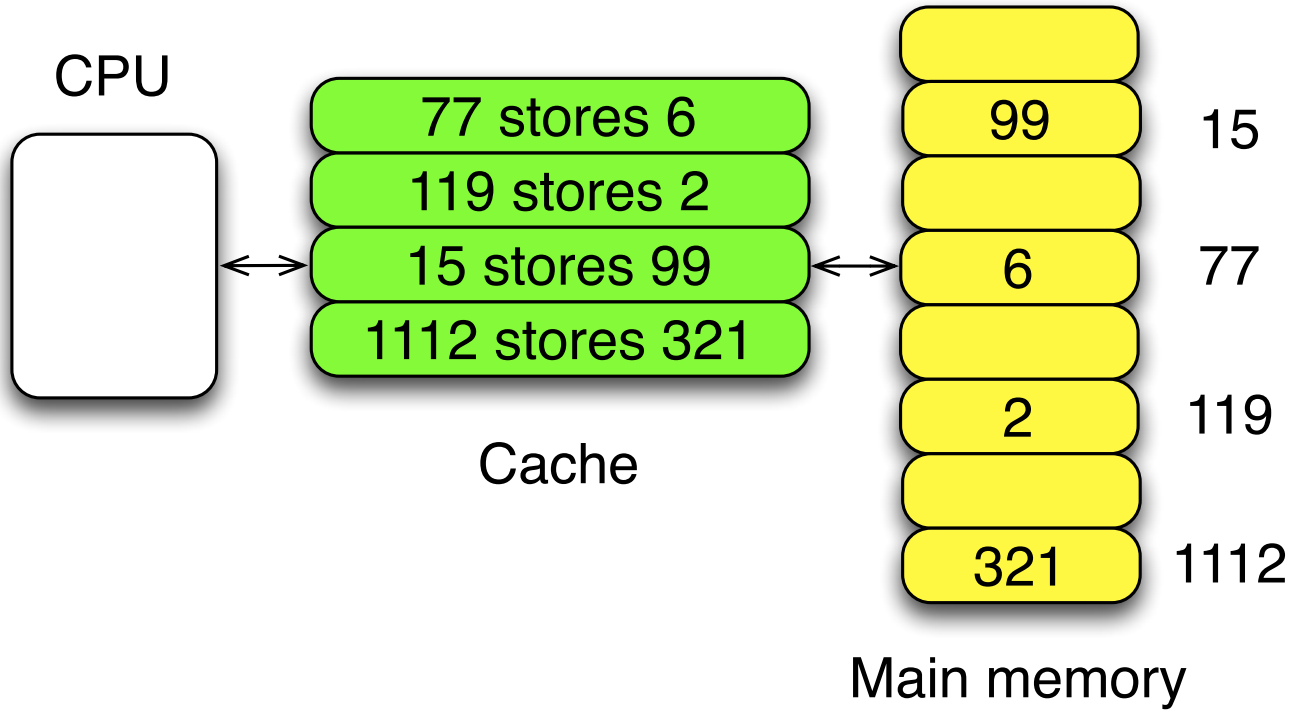
Cache

A CPU **cache** is used by the CPU of a computer to reduce the average time to access memory. The cache is a smaller, faster and more expensive memory inside the CPU which stores copies of the data from the **most frequently** used main memory locations for fast access. A cache entry is called **cache line**.

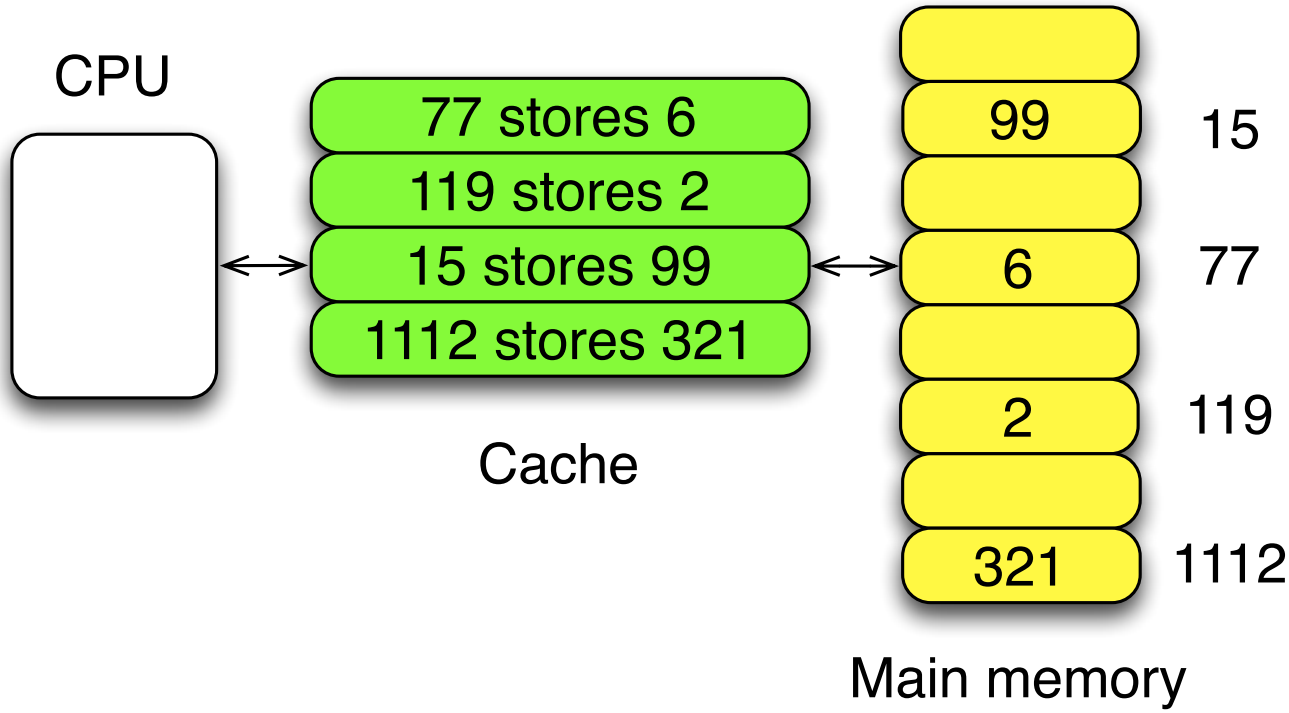
When the CPU reads from or writes to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory.

Recall that **latency** (of memory access) is the time between issuing the read/write access and the completion of this access.

Cache (highly simplified)

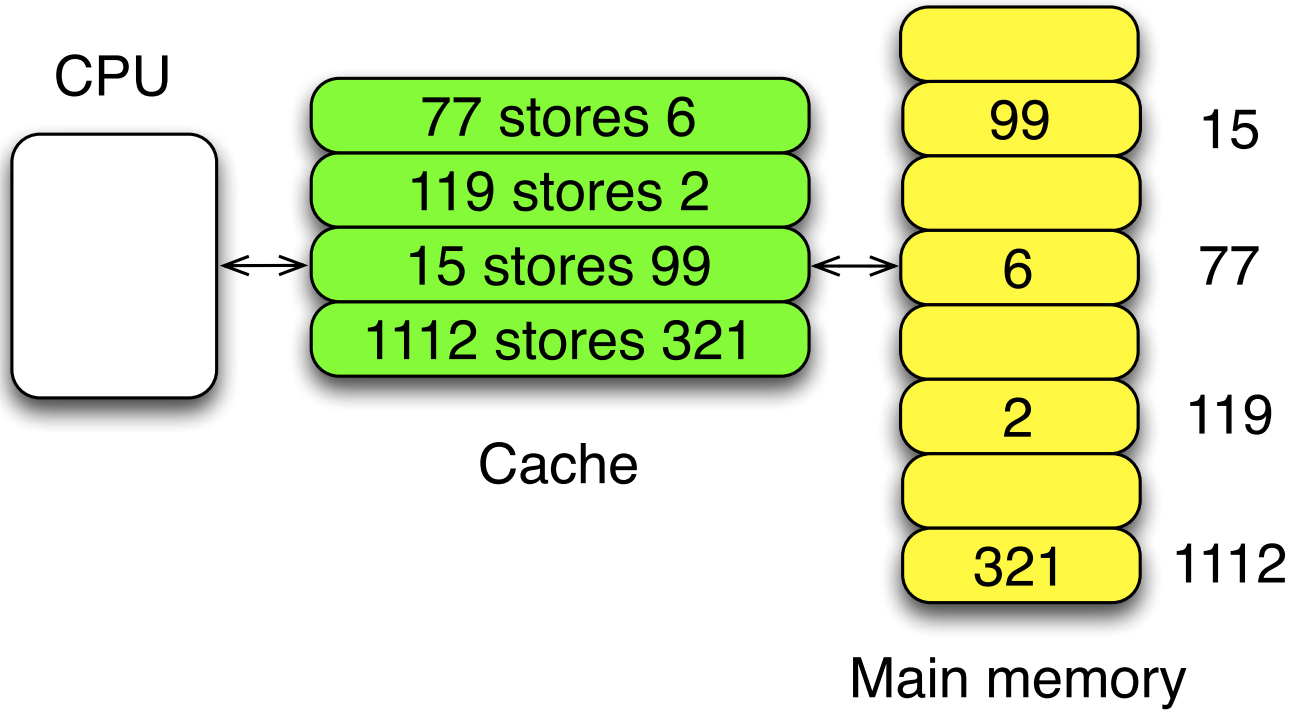


Cache (highly simplified)



Cache contains **temporary copies** of selected main memory locations, eg. Mem[119] = 2.

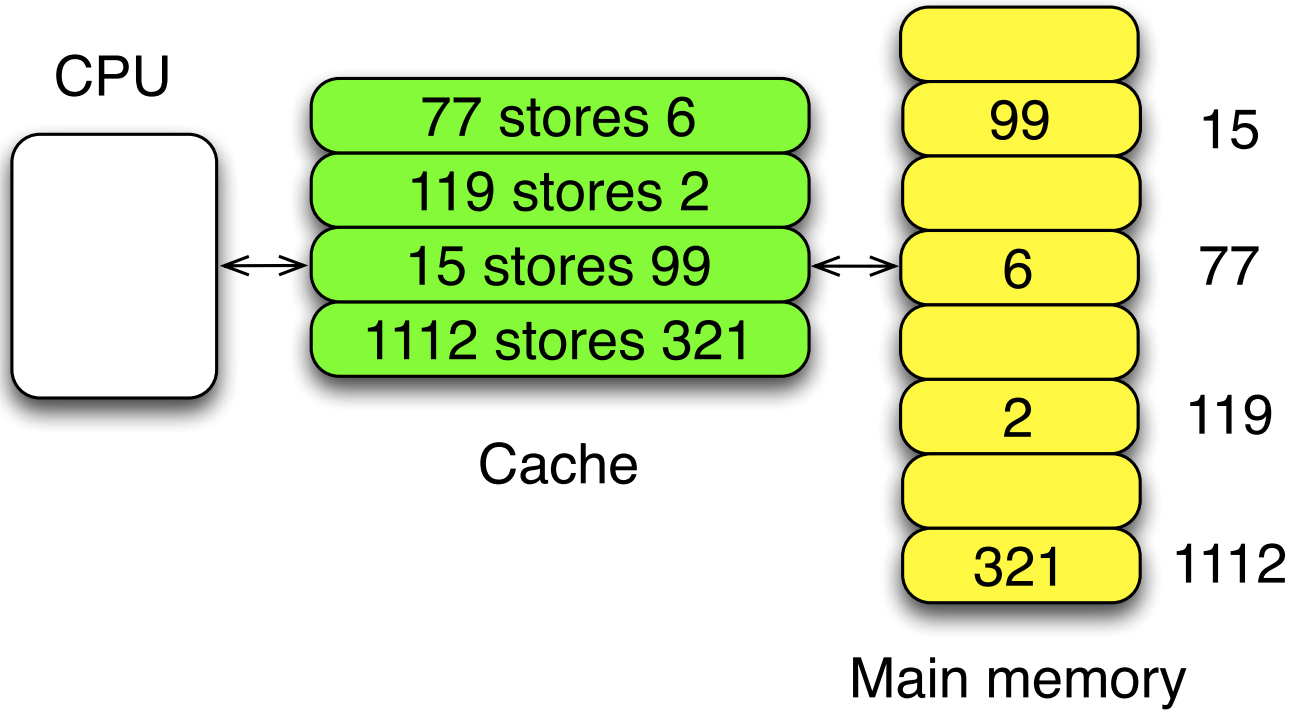
Cache (highly simplified)



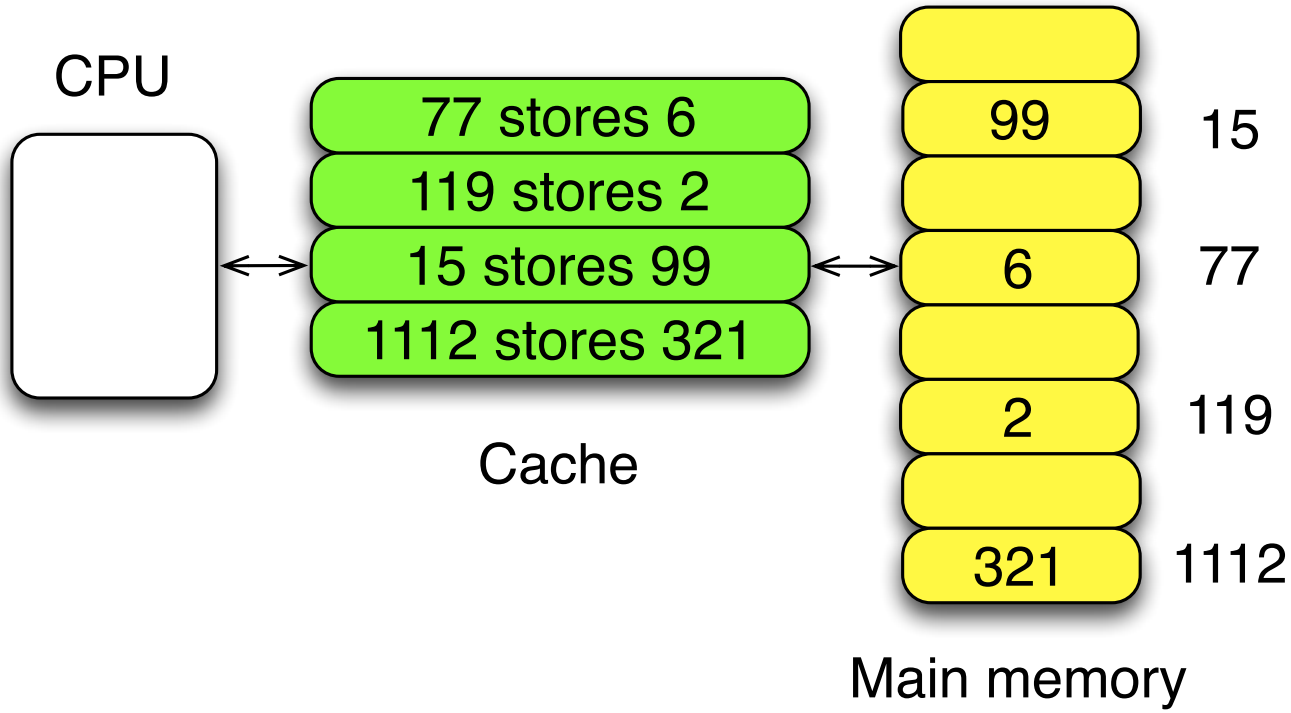
Cache contains **temporary copies** of selected main memory locations, eg. Mem[119] = 2.

The cache holds **pairs** of main memory address (called **tag**) and value.

Cache (highly simplified)

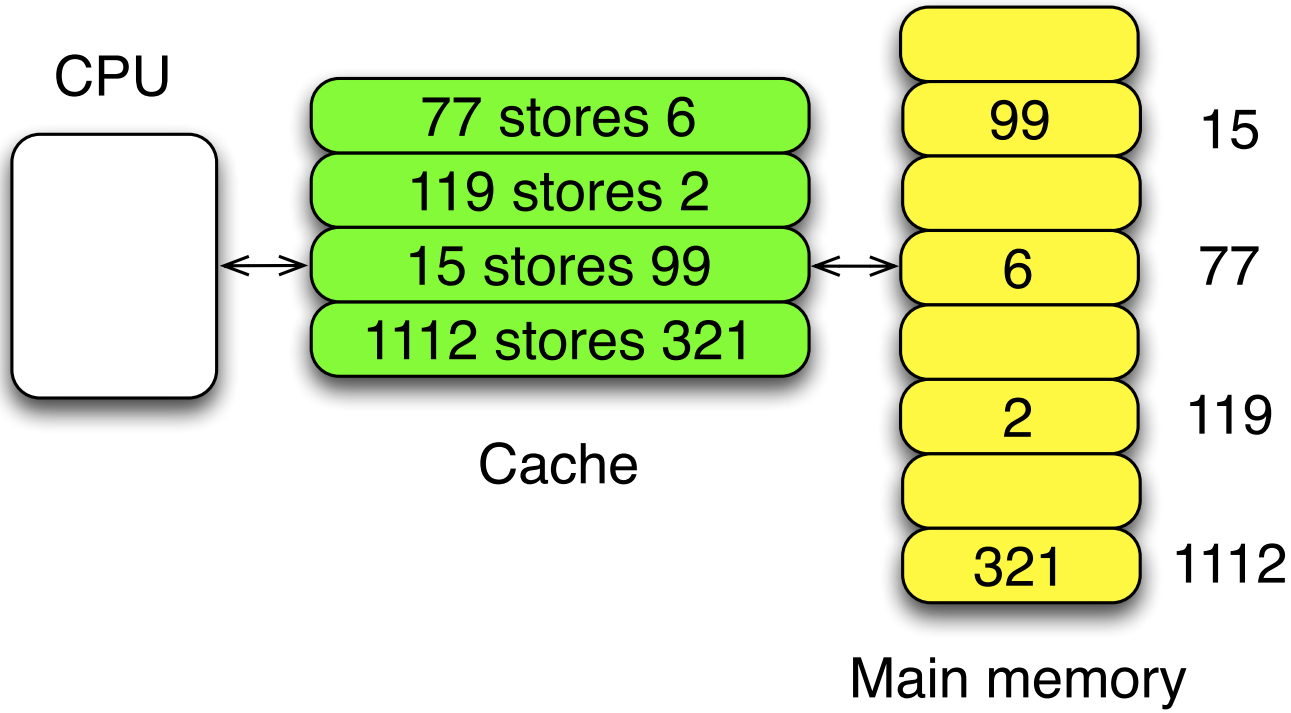


Cache (highly simplified)



Goal of a cache: to reduce the **average** access time to memory by exploiting locality.

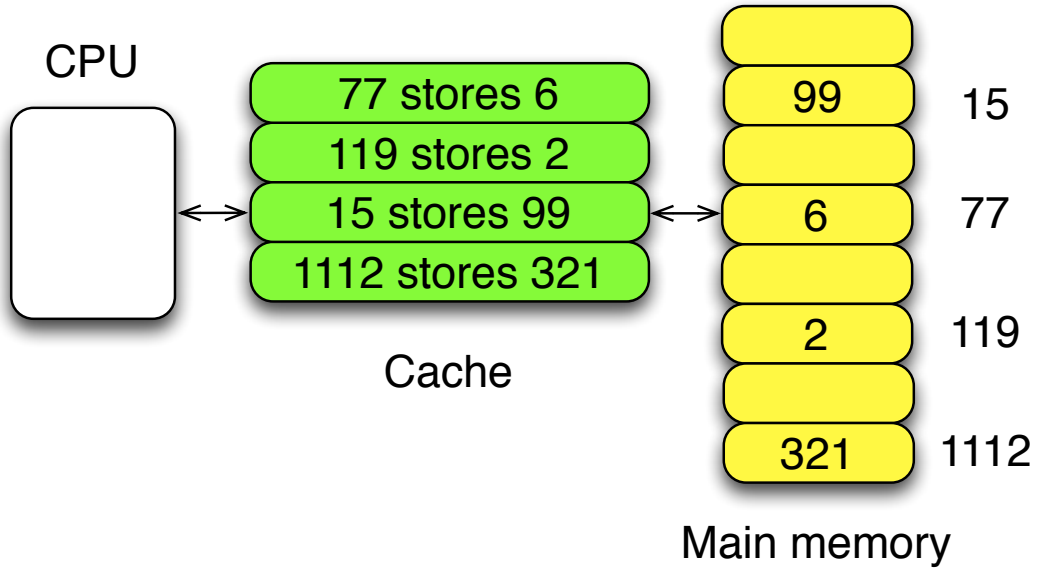
Cache (highly simplified)



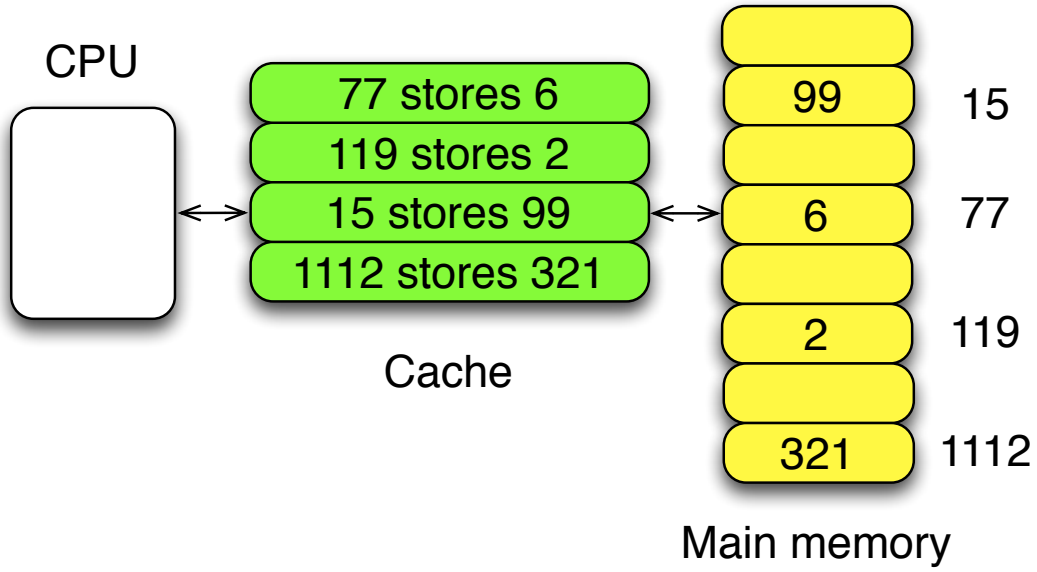
Goal of a cache: to reduce the **average** access time to memory by exploiting locality.

Caches are made from expensive but fast SRAM, with much less capacity than main memory. So not all memory entries can be in cache.

Cache reading (highly simplified)



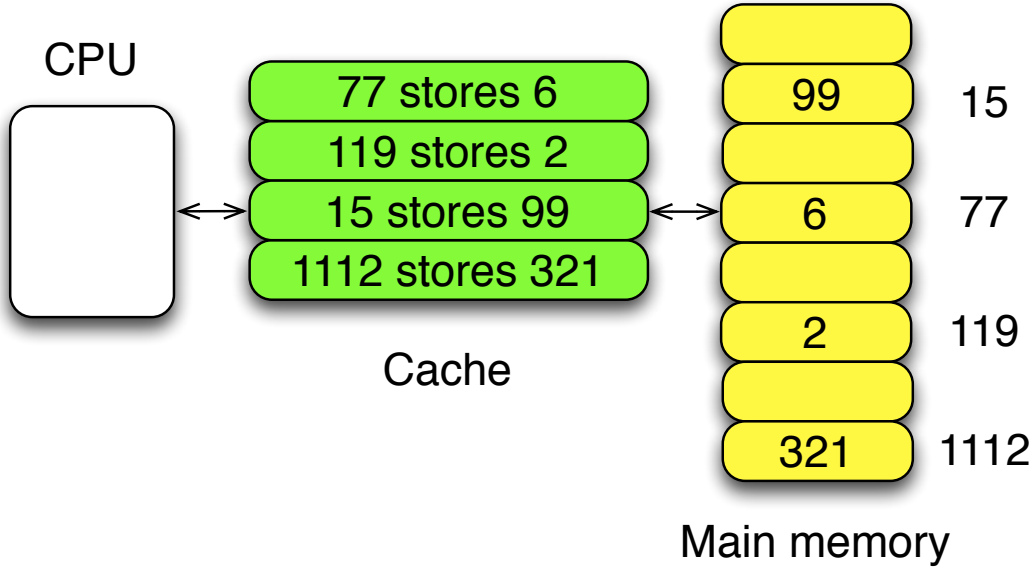
Cache reading (highly simplified)



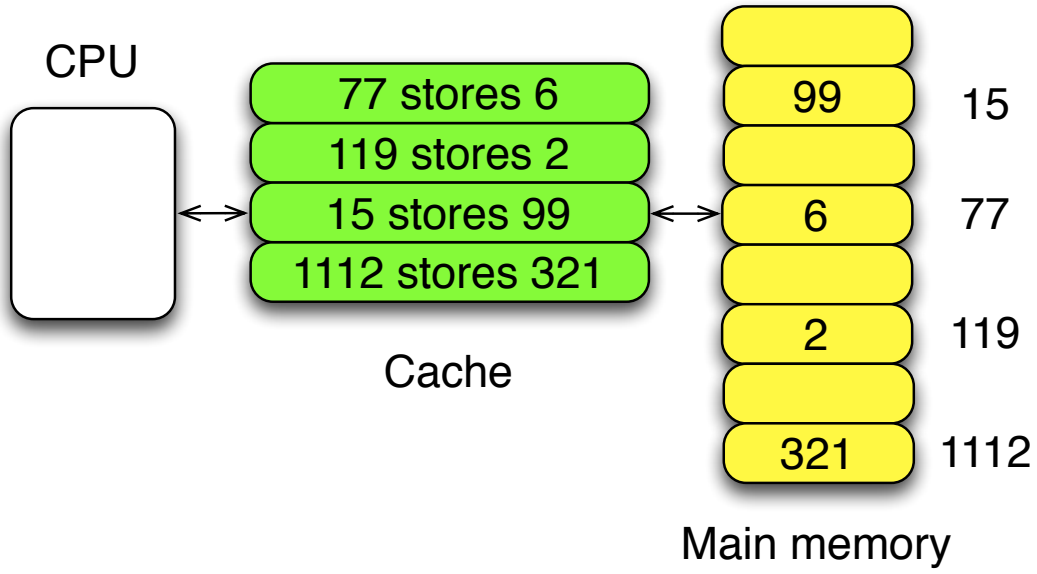
Cache 'algorithm': if the CPU wants to read memory location `loc`:

- Look for tag `loc` in cache.
- If cache line (`loc`, val) is found (called **cache hit**), then return val.
- If no cache line contains tag `loc` (called **cache miss**), then select some cache line k for replacement, read location `loc` from main memory getting value val', replace k with (`loc`, val').

Cache writing (highly simplified)



Cache writing (highly simplified)



Cache 'algorithm': if the CPU wants to write the value val memory location `loc`:

- Look for tag `loc` in cache.
- If cache line (`loc`, val') is found (**cache hit**), then replace val' with val in the cache line.
- If no cache line contains tag `loc` (**cache miss**), then select some cache line k for replacement, replace k with (`loc`, val).
- Write val to main memory location `loc`.

Note

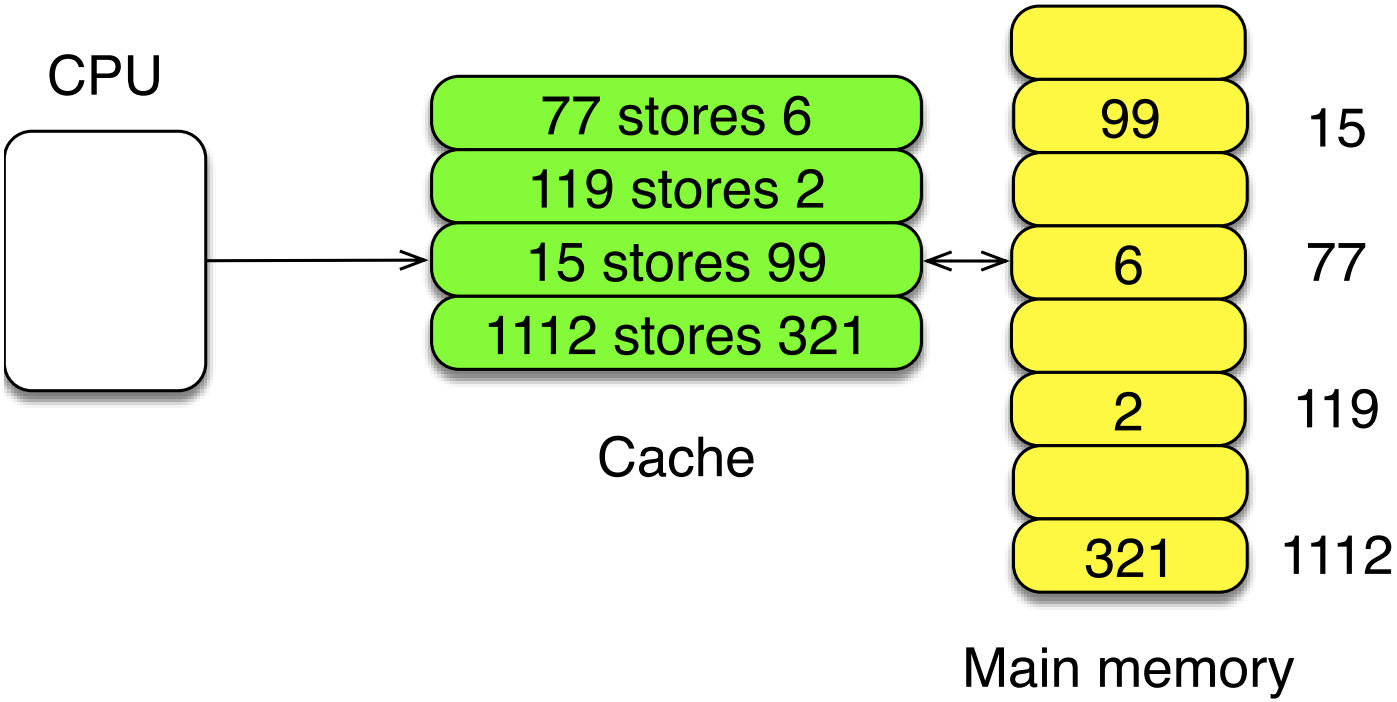
All these things (writing to main memory, looking for tag, replacing cache line, evicting etc) happen **automatically**, behind the programmer's back. It's all implemented in silicon, so cache management is **very fast**.

Note

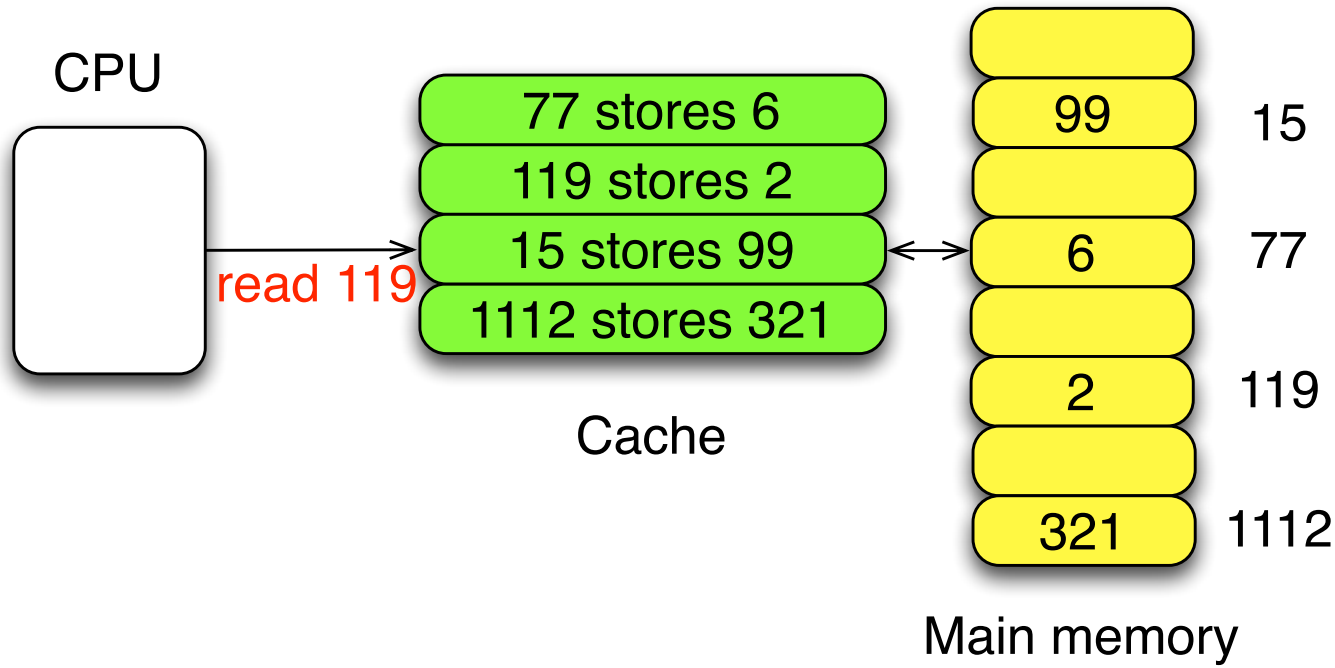
All these things (writing to main memory, looking for tag, replacing cache line, evicting etc) happen **automatically**, behind the programmer's back. It's all implemented in silicon, so cache management is **very fast**.

Unless interested in peak performance, the programmer can program under the illusion of memory uniformity.

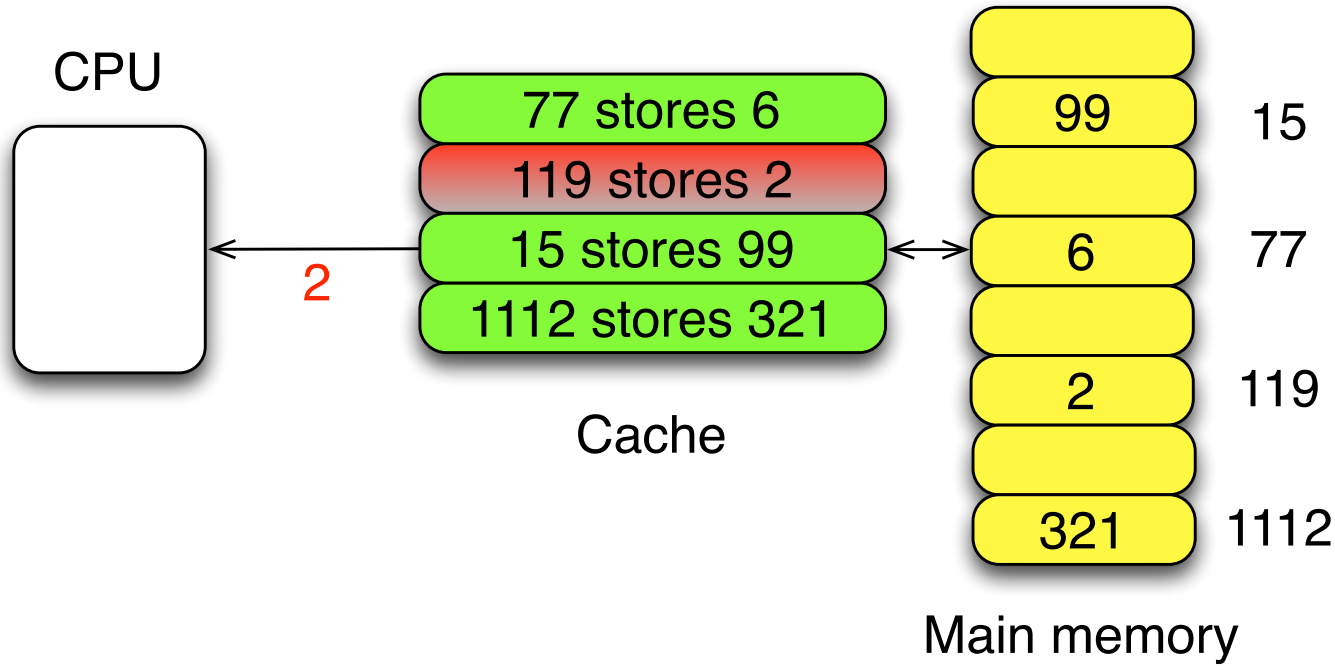
Successful cache read (highly simplified)



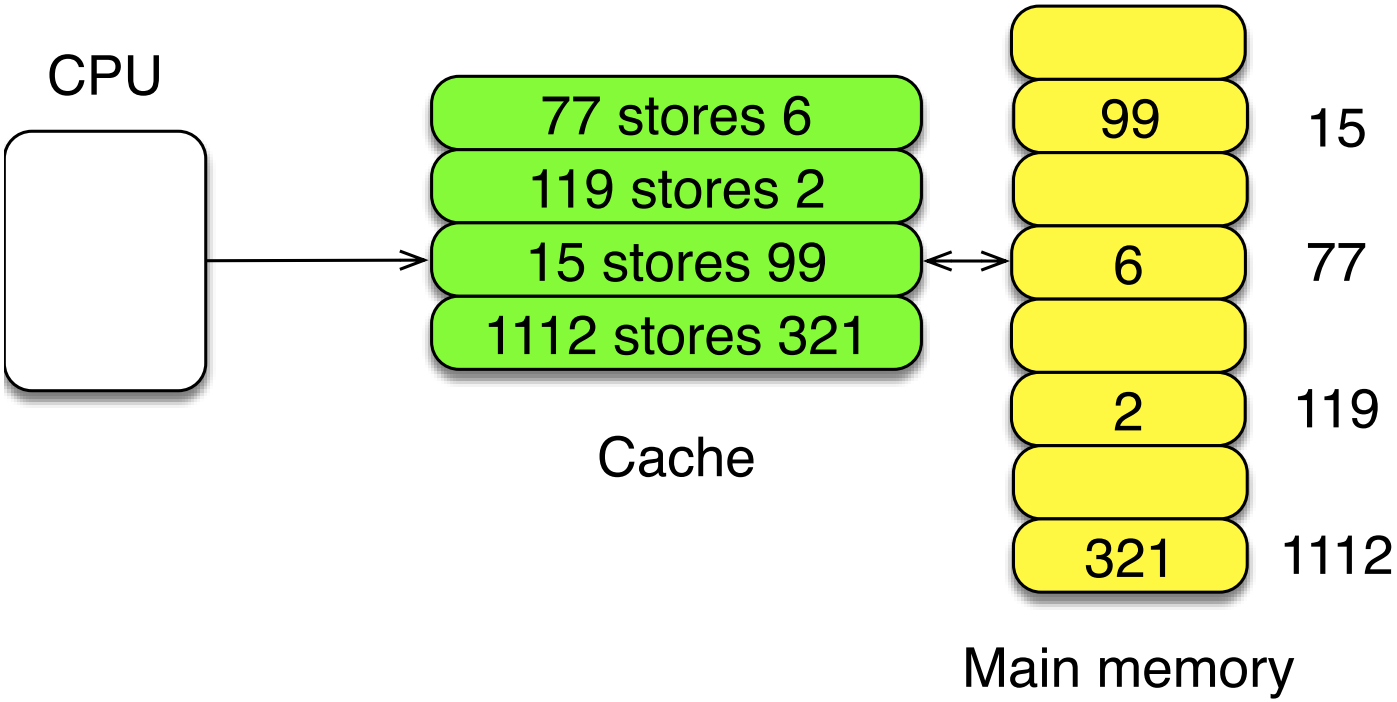
Successful cache read (highly simplified)



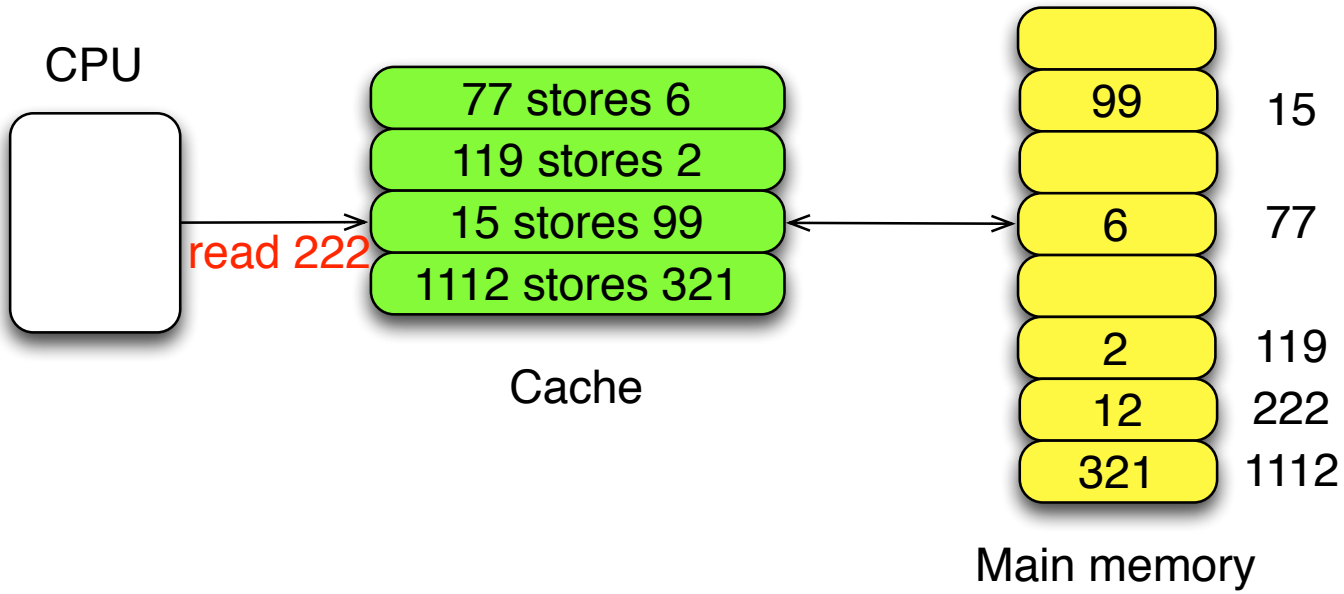
Successful cache read (highly simplified)



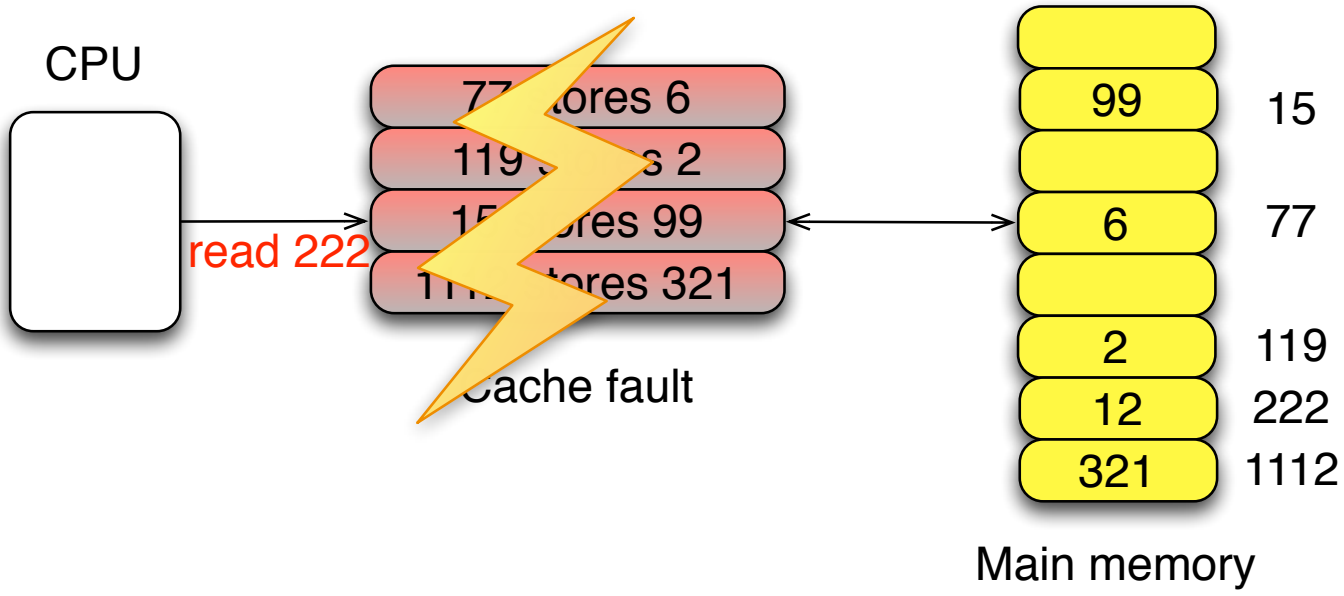
Successful cache read (highly simplified)



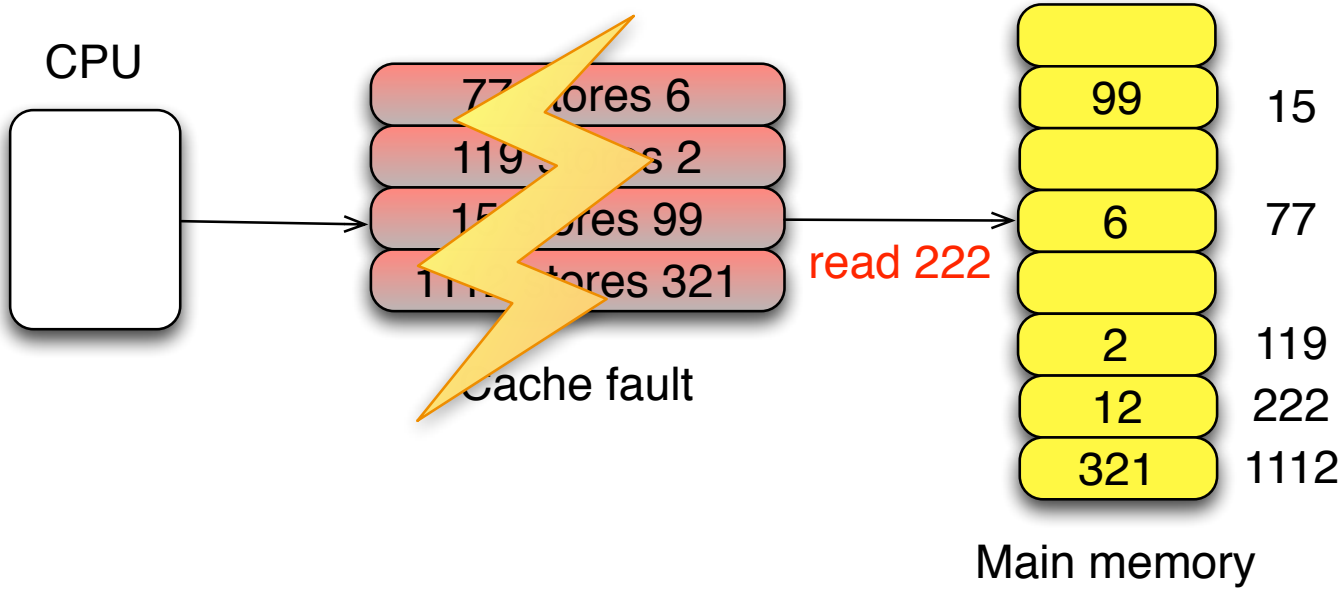
Cache failure on read (highly simplified)



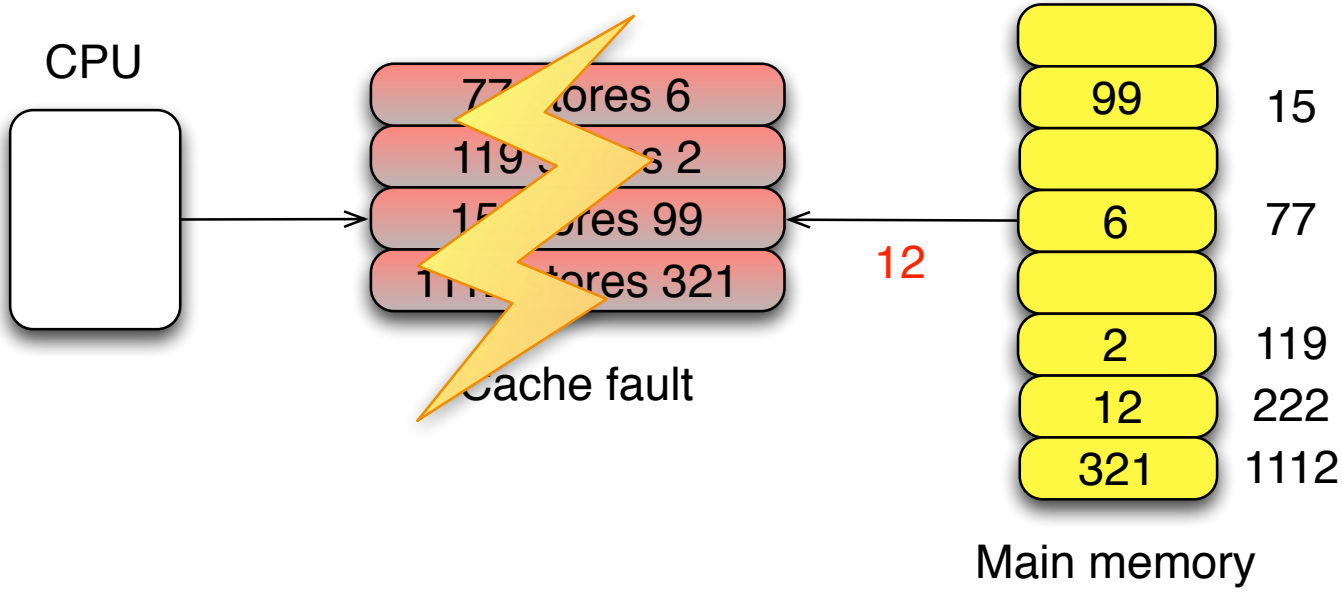
Cache failure on read (highly simplified)



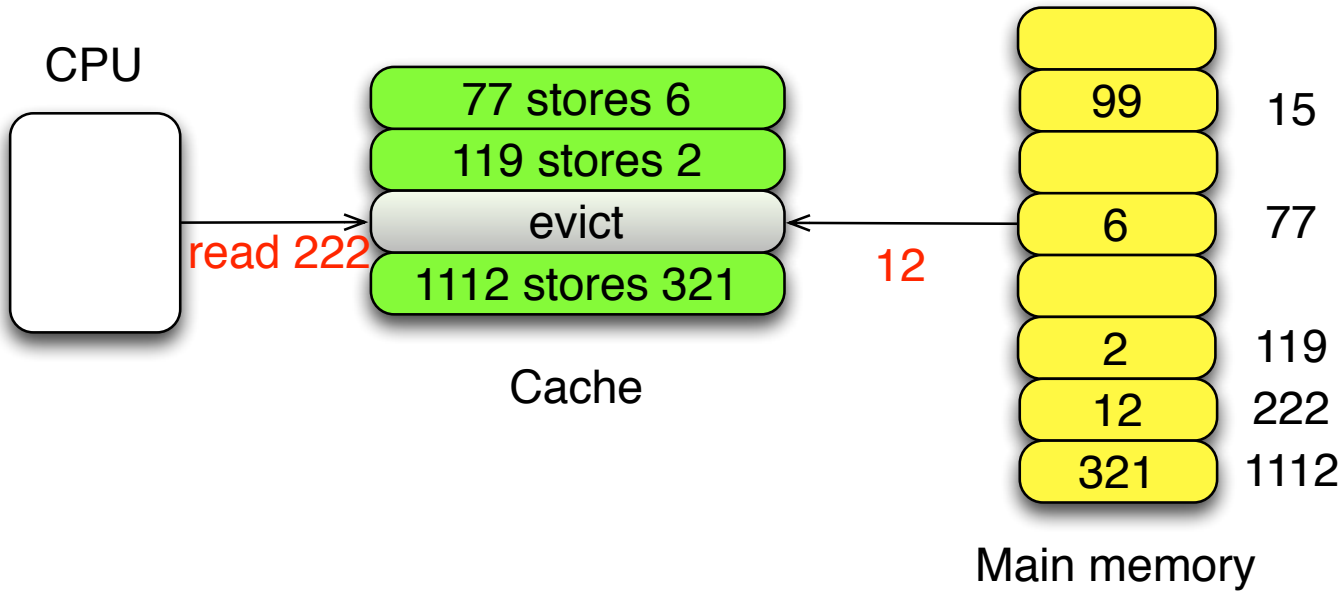
Cache failure on read (highly simplified)



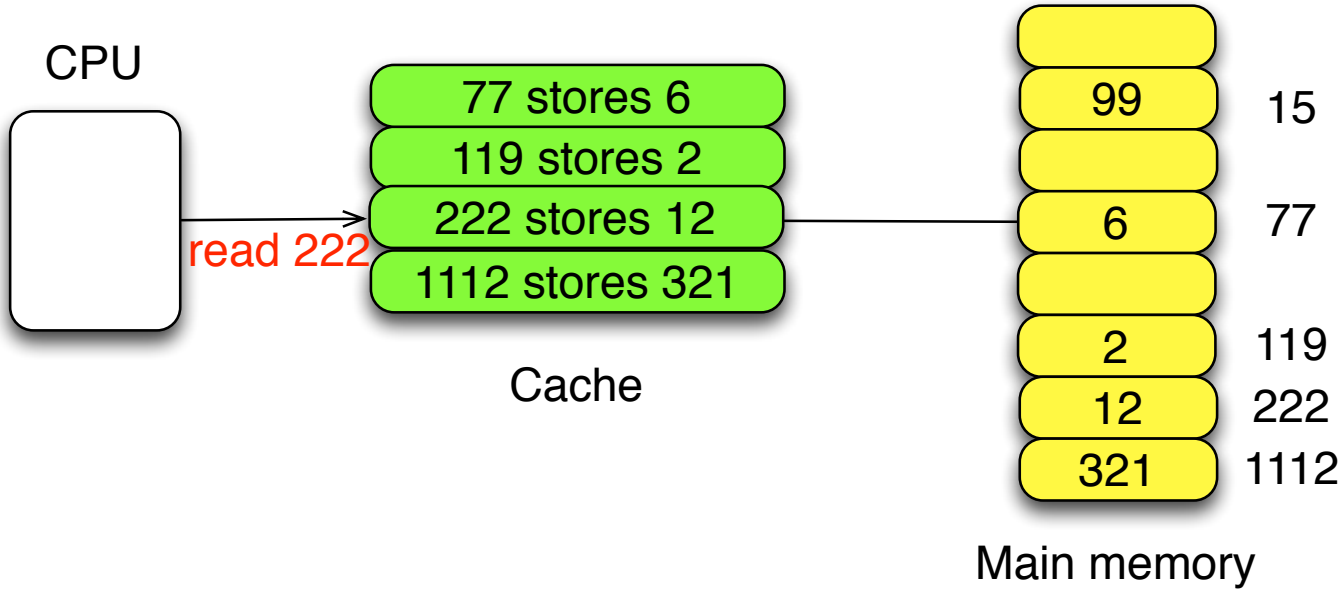
Cache failure on read (highly simplified)



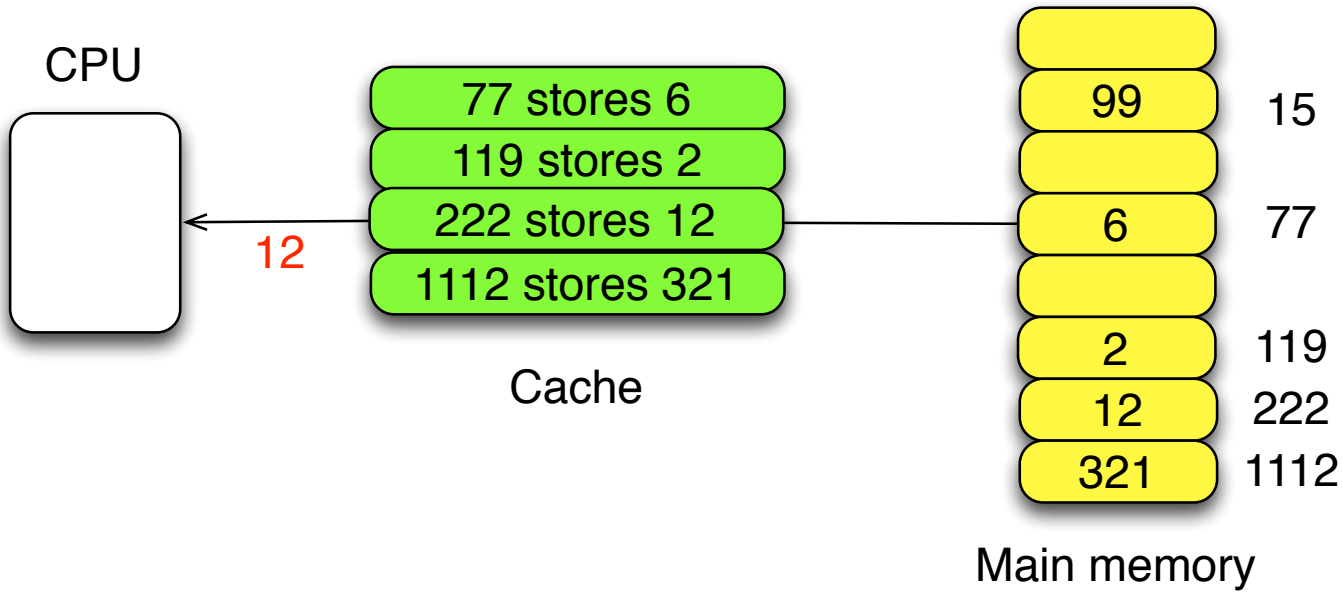
Cache failure on read (highly simplified)



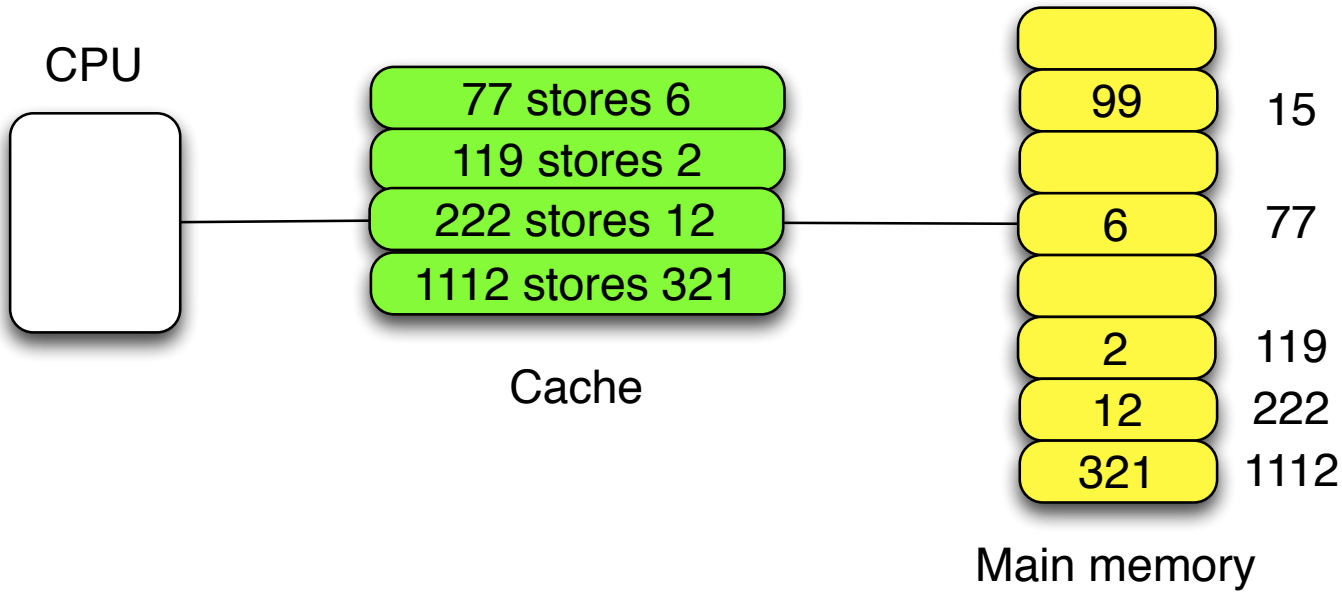
Cache failure on read (highly simplified)



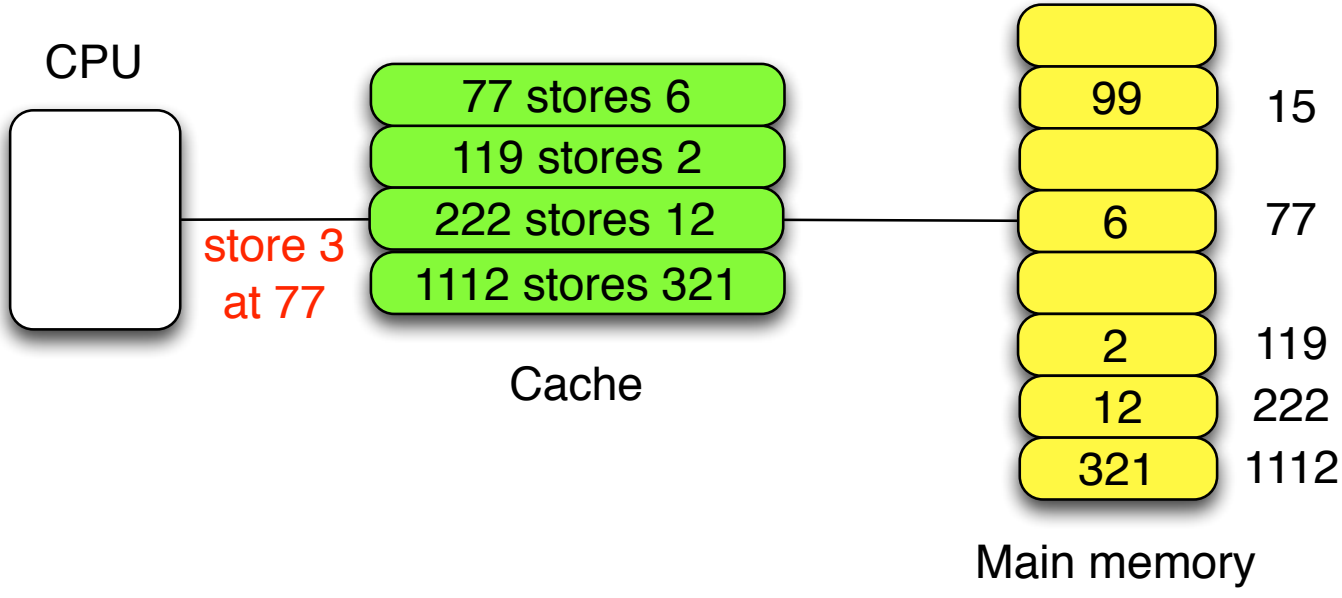
Cache failure on read (highly simplified)



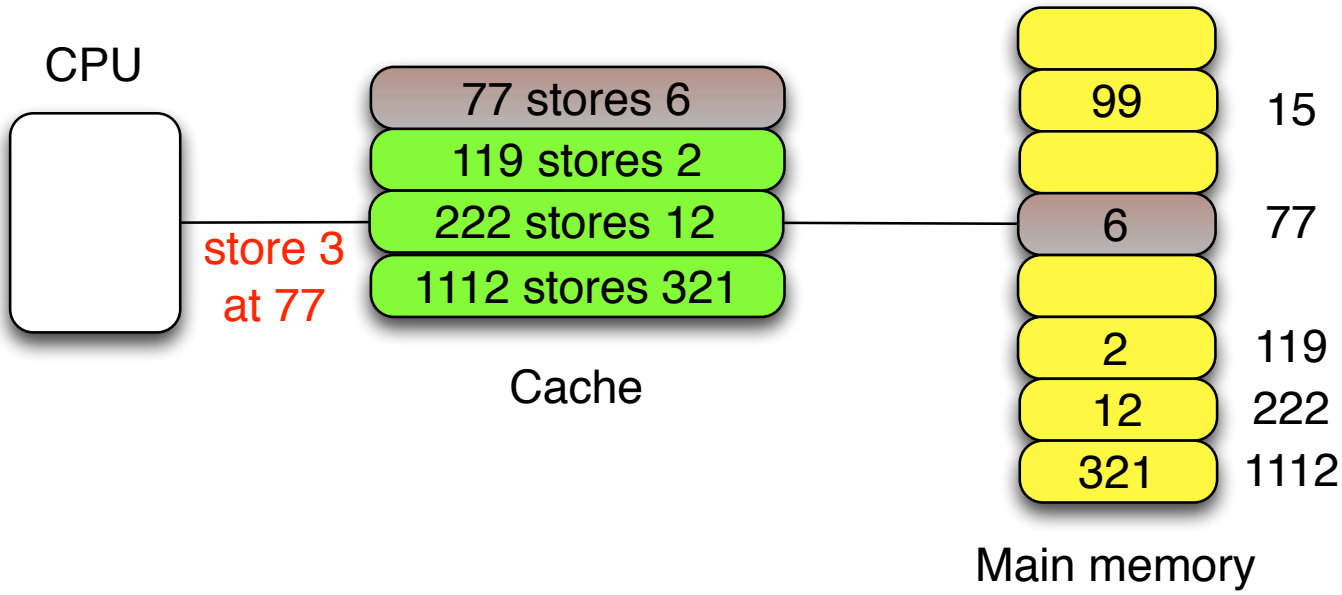
Cache failure on read (highly simplified)



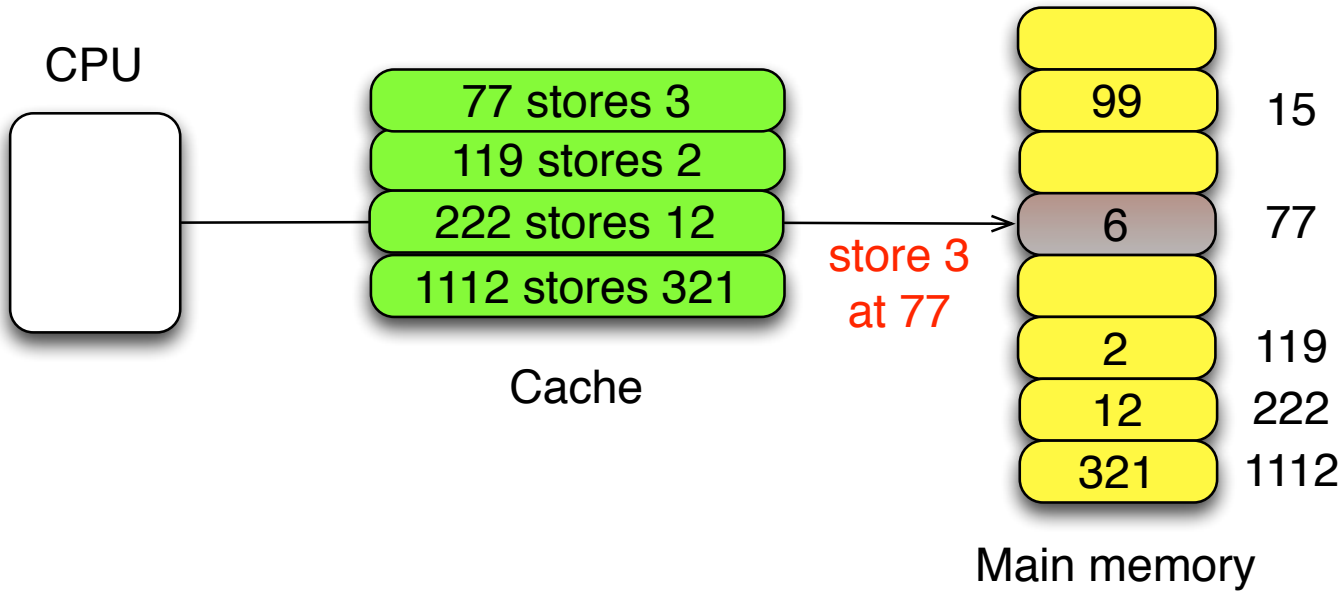
Successful cache write (highly simplified)



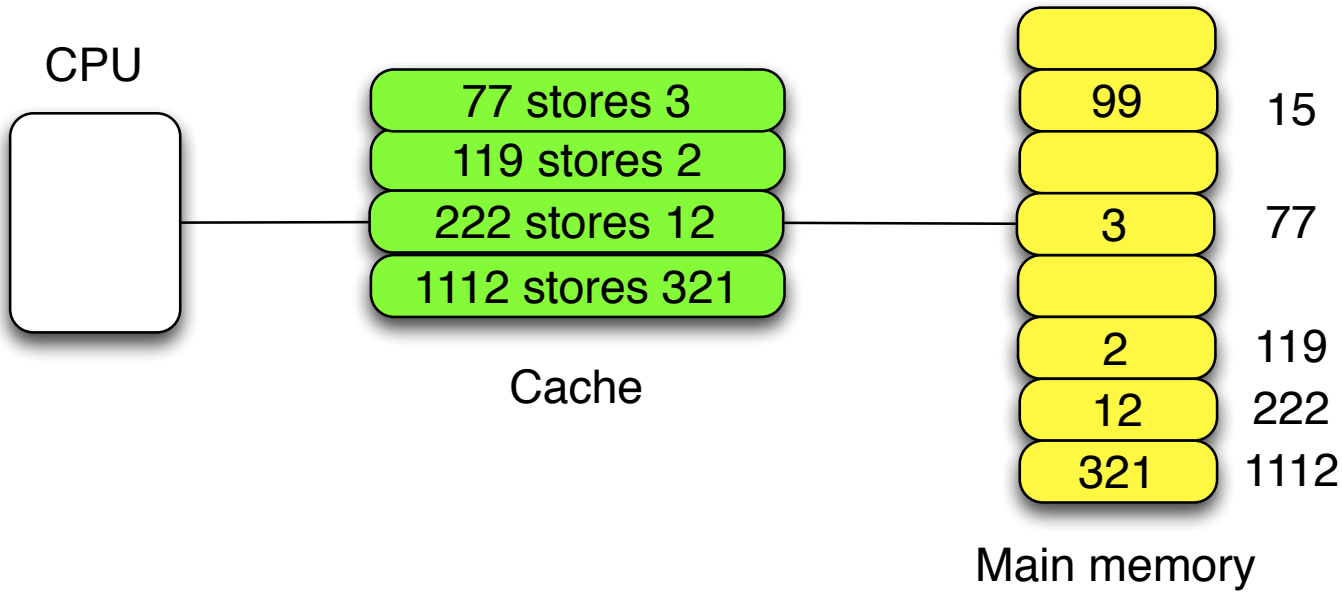
Successful cache write (highly simplified)



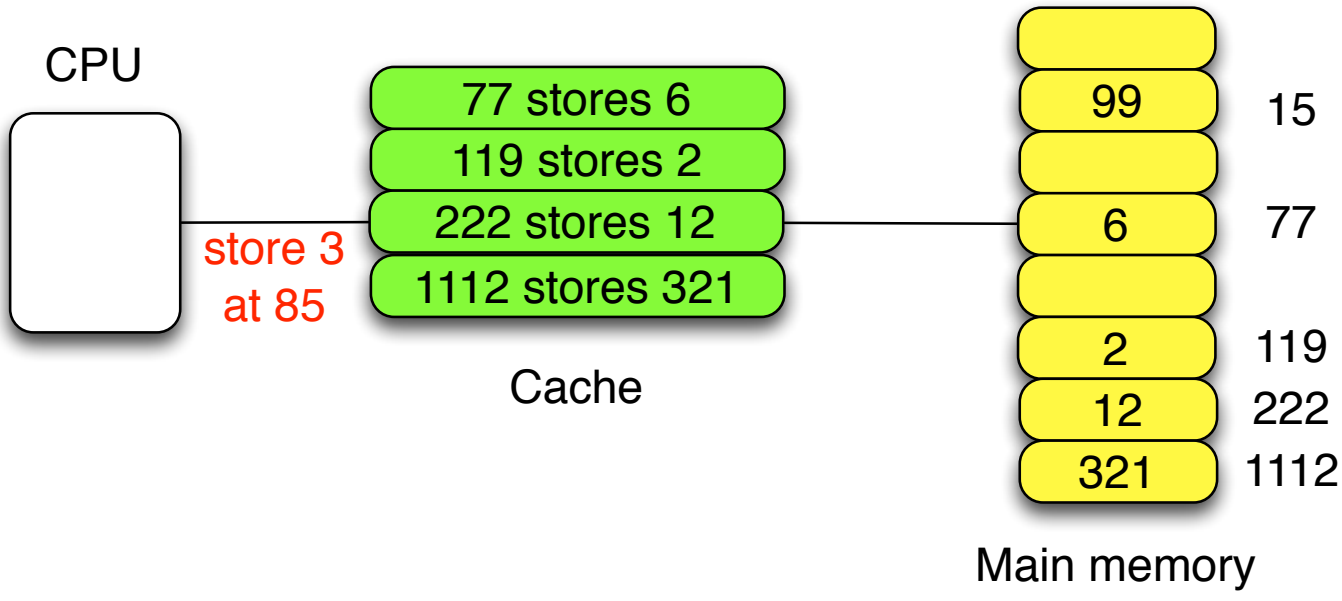
Successful cache write (highly simplified)



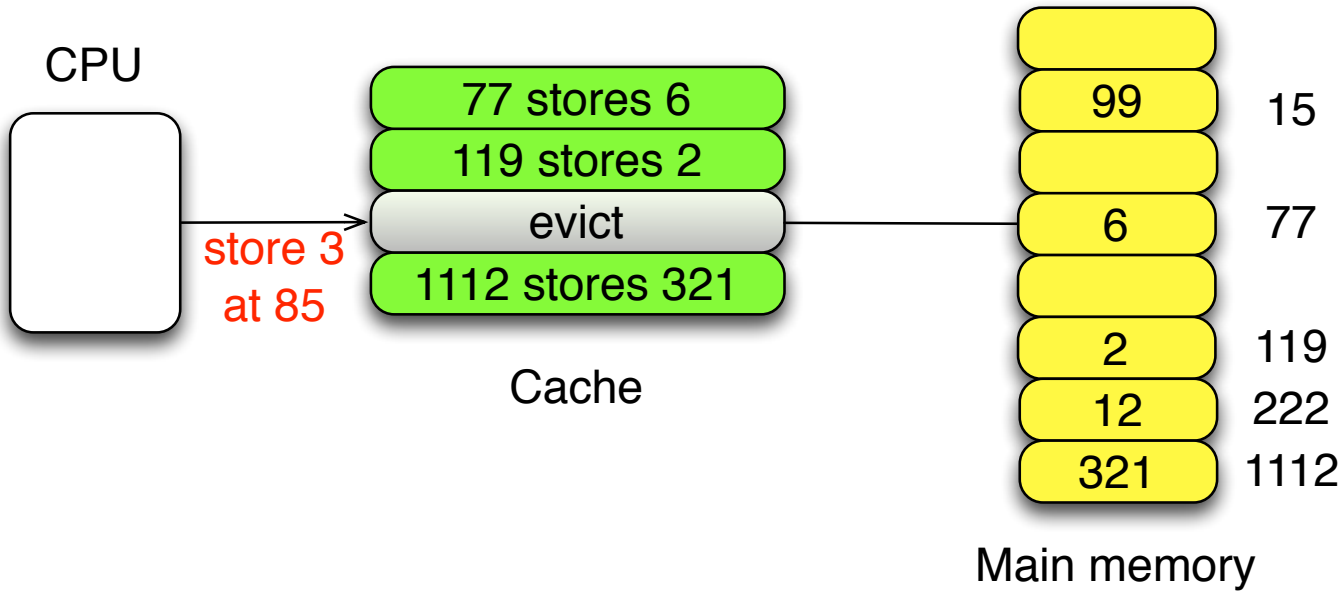
Successful cache write (highly simplified)



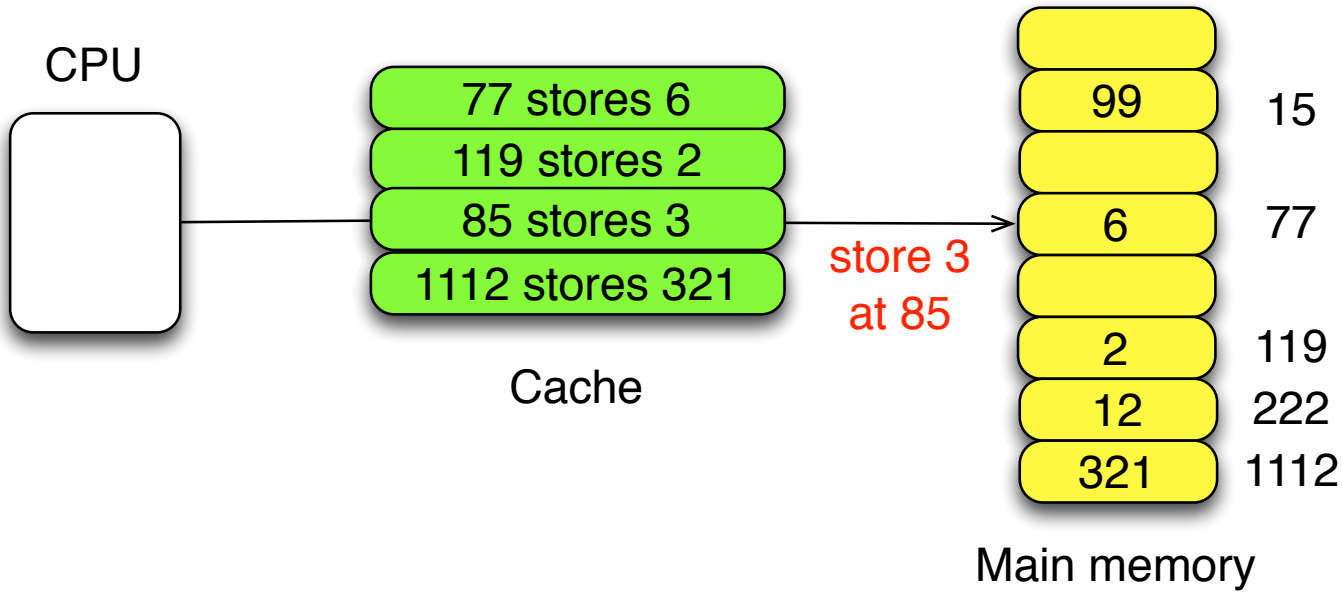
Write with cache failure (highly simplified)



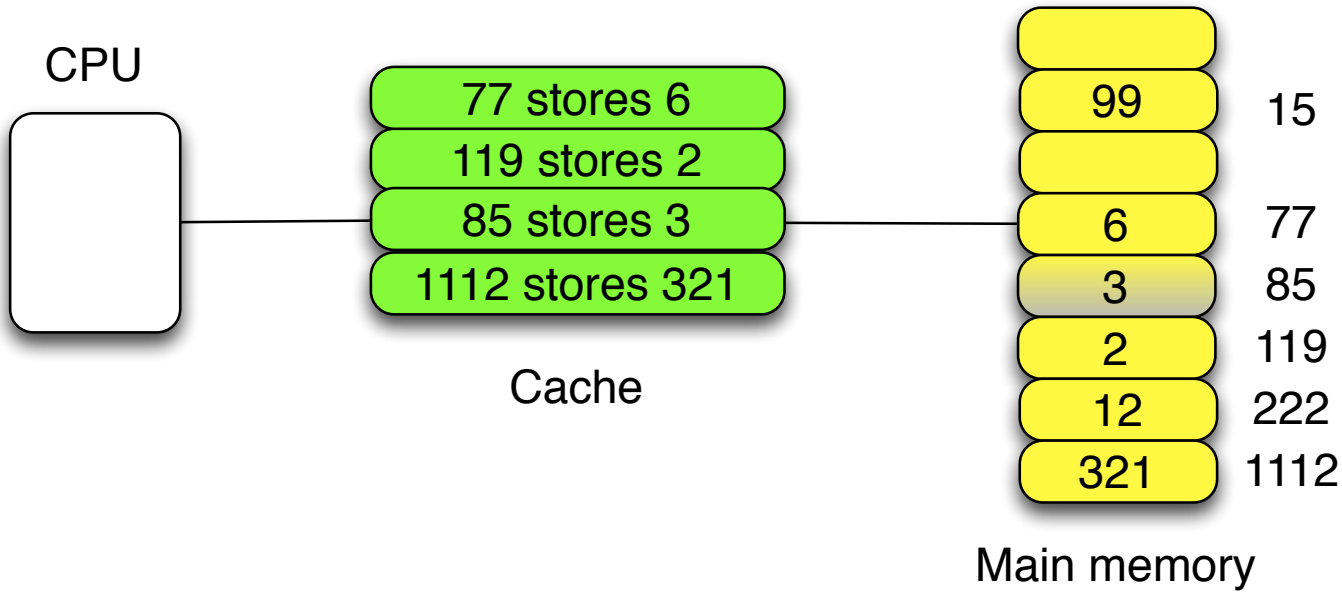
Write with cache failure (highly simplified)



Write with cache failure (highly simplified)



Write with cache failure (highly simplified)



Prefetching

How is this related to locality?

Prefetching

How is this related to locality?

By locality, the act of accessing `loc` predicts likely usage of

```
... loc-1   loc   loc+1   loc+2   loc+3   ...
```

in the near future. So far we are only 'reusing' `loc`.

Prefetching

How is this related to locality?

By locality, the act of accessing `loc` predicts likely usage of

```
... loc-1   loc   loc+1   loc+2   loc+3   ...
```

in the near future. So far we are only 'reusing' `loc`.

So let's get e.g. `loc...loc+n` from memory together (rather than separately). This is called **prefetching** . It relies on the fact that with current CPUs and memory systems a burst of reads of *n* adjacent addresses from main memory is faster than fetching those *n* cells separately.

Prefetching

How is this related to locality?

By locality, the act of accessing `loc` predicts likely usage of

```
... loc-1   loc   loc+1   loc+2   loc+3   ...
```

in the near future. So far we are only 'reusing' `loc`.

So let's get e.g. `loc...loc+n` from memory together (rather than separately). This is called **prefetching** . It relies on the fact that with current CPUs and memory systems a burst of reads of *n* adjacent addresses from main memory is faster than fetching those *n* cells separately.

The concept of **preloading** of webpages in is similar.
(cf. Instagram)

Counterintuitive behaviour of CPUs with caches

Consider the following simple program.

```
x := x - y;
y := y - x
```

Clearly both assignments translate to very similar assembly code. But if the CPU has caches (and all modern CPUs have), then the execution of the first execution of `x := x - y` might take much longer than the execution of the second `y := y - x`, despite being very similar. Why?

Counterintuitive behaviour of CPUs with caches

Consider the following simple program.

```
x := x - y;
y := y - x
```

Clearly both assignments translate to very similar assembly code. But if the CPU has caches (and all modern CPUs have), then the execution of the first execution of `x := x - y` might take much longer than the execution of the second `y := y - x`, despite being very similar. Why?

Because in the first execution of `x := x - y`, the cache may not contain `x` or `y`. In this case, a slow request will be issued to main memory to fetching `x`, or `y`, or both. Thus, the second statement is likely to execute more quickly, because the cache will contain both `x` and `y`.

Miss rate

$$\text{Miss rate} = \frac{\text{number of cache failures}}{\text{number of cache accesses}}.$$

Miss rate

$$\text{Miss rate} = \frac{\text{number of cache failures}}{\text{number of cache accesses}}.$$

The performance of a program can be significantly improved if the miss rate is low.

Miss rate

$$\text{Miss rate} = \frac{\text{number of cache failures}}{\text{number of cache accesses}}.$$

The performance of a program can be significantly improved if the miss rate is low.

Miss rate is influenced by many factors:

Miss rate

$$\text{Miss rate} = \frac{\text{number of cache failures}}{\text{number of cache accesses}}.$$

The performance of a program can be significantly improved if the miss rate is low.

Miss rate is influenced by many factors:

- Selection policy for cache eviction.
- Compiler improving data locality (e.g. in garbage collection).
- Programmer ensuring data locality.
- Cache size (bigger = better). Modern computers have multiple caches (see later).

Miss penalty

The **miss penalty** of a cache are the time it takes to read from/write to main memory - (minus) the time it takes to read from/write to the cache.

Miss penalty

The **miss penalty** of a cache are the time it takes to read from/write to main memory - (minus) the time it takes to read from/write to the cache.

The miss penalty has been increasing dramatically, because CPUs get faster more quickly than memory. So good cache management is becoming increasingly important.

Important questions about Cache

Important questions about Cache

How does the search in the cache for a tag proceed?

Important questions about Cache

How does the search in the cache for a tag proceed?

Which cache line is replaced after a cache miss?

Important questions about Cache

How does the search in the cache for a tag proceed?

Which cache line is replaced after a cache miss?

When to update main memory after cache write?

Important questions about Cache

How does the search in the cache for a tag proceed?

Which cache line is replaced after a cache miss?

When to update main memory after cache write?

What (how much) data to read from main memory after a cache miss?

Important questions about Cache

How does the search in the cache for a tag proceed?

Which cache line is replaced after a cache miss?

When to update main memory after cache write?

What (how much) data to read from main memory after a cache miss?

The answers to both are vitally important for performance. Different CPUs give different answers. Modern CPUs are **very** sophisticated in these matters: good answers have dramatic impact on performance.

Important questions about Cache

Important questions about Cache

Example: which cache line is replaced after a cache miss?

Important questions about Cache

Example: which cache line is replaced after a cache miss?
Different policies:

- Random choice.
- Least recently used.
- Most recently used.
- FIFO.
- LIFO
- ...

See e.g. `https://en.wikipedia.org/wiki/Cache_replacement_policies` for more.

Important questions about Cache

Important questions about Cache

Example: when to update main memory after cache write?

Important questions about Cache

Example: when to update main memory after cache write?

Different policies:

- Immediately on write. This is simple and simple to implement, but can potentially stall the CPU while being executed.
- Later, e.g. when there is no/little contention for memory bus. This leads to strange effects from the programmer's point of view, where seemingly instructions are **reordered**, e.g. the programmer says: write this, then read that, but the CPU executes: read that then write this. This reordering of instructions is called the CPU's **memory model**. Modern CPUs have complicated memory models. Compiler writes and assembler programmers need to be aware of this, otherwise programs will be buggy. 'Normal' programmers can (fortunately) ignore this.

Important questions about Cache

Important questions about Cache

Example: what (how much) data to read from main memory after a cache miss?

Important questions about Cache

Example: what (how much) data to read from main memory after a cache miss?

Different CPUs give different answers. Due to locality, it makes sense to load `loc+0`, ..., `loc+n` on cache fault for location `loc`. Modern CPUs typically determine *n* **dynamically**.

Important questions about Cache

Important questions about Cache

What do real processors use?

Important questions about Cache

What do real processors use? That's quite hard to say, but looks complicated. CPU producers seem to keep this secret. If you know more, let me know ...

Important questions about Cache

What do real processors use? That's quite hard to say, but looks complicated. CPU producers seem to keep this secret. If you know more, let me know ...

Why can they keep this secret?

Important questions about Cache

What do real processors use? That's quite hard to say, but looks complicated. CPU producers seem to keep this secret. If you know more, let me know ...

Why can they keep this secret? Because caching does not affect the results programs compute, only speed. Give manufacturers freedom to change CPU architecture without telling anyone.

Problem

Modern compilers are very good at managing registers, much better than almost all programmers could.

Problem

Modern compilers are very good at managing registers, much better than almost all programmers could.

Compilers are **not** good at managing caches. This problem is left to the programmer. It is an open research question of to make compilers better at managing caches.

Problem

Modern compilers are very good at managing registers, much better than almost all programmers could.

Compilers are **not** good at managing caches. This problem is left to the programmer. It is an open research question of to make compilers better at managing caches.

One problem: OS switches between processes, and a context switch typically 'trashes' the cache, i.e. the cache holds values that are good for the outgoing process, but hold no values of interest to the incoming process.

Multi-level caches

Multi-level caches

The picture painted about caches so far is too simplistic in that modern CPUs have not one but (usually) three caches:

- L1. Small, very fast.
- L2. Medium size, medium speed.
- L3. Large size, slow (but still much faster than main memory).

(L stands for Level.) Intel has just started adding L4 caches (Haswell)

Multi-level caches

The picture painted about caches so far is too simplistic in that modern CPUs have not one but (usually) three caches:

- L1. Small, very fast.
- L2. Medium size, medium speed.
- L3. Large size, slow (but still much faster than main memory).

(L stands for Level.) Intel has just started adding L4 caches (Haswell)

Processor	L1	L2	L3
ARM Coretex A9	32 KB	128kB-8MB	-
ARM Coretex A15	64 KB	4MB	-
Intel Ivy Bridge	64 KB p. core	256 kB p. core	8 MB shared

Multi-level caches

The picture painted about caches so far is too simplistic in that modern CPUs have not one but (usually) three caches:

- L1. Small, very fast.
- L2. Medium size, medium speed.
- L3. Large size, slow (but still much faster than main memory).

(L stands for Level.) Intel has just started adding L4 caches (Haswell)

Processor	L1	L2	L3
ARM Coretex A9	32 KB	128kB-8MB	-
ARM Coretex A15	64 KB	4MB	-
Intel Ivy Bridge	64 KB p. core	256 kB p. core	8 MB shared

Why?

Multi-level caches

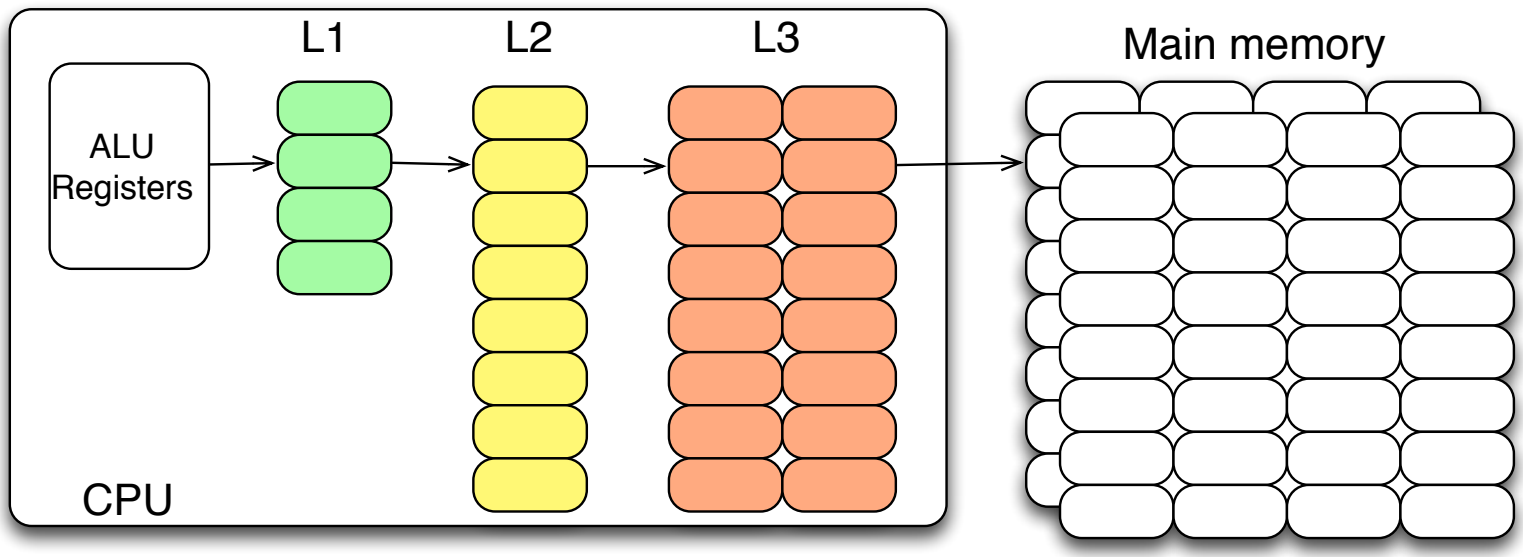
There is a fundamenal and hard trade-off in caches:

- Smaller chaches are faster, have lower latency.
- Bigger caches have a better miss-rate.

It seems we can't have both: fast caches and low miss-rate.

Multi-level caches

To deal with the miss-rate/low latency trade-off, modern CPU create a **hierarchy** of caches: the small but fast L1 cache doesn't read directly from memory but from a bigger but slower L2 cache. In turn the L2 cache often reads from a even larger and even more slow L3 cache. The L3 cache reads from the main memory.

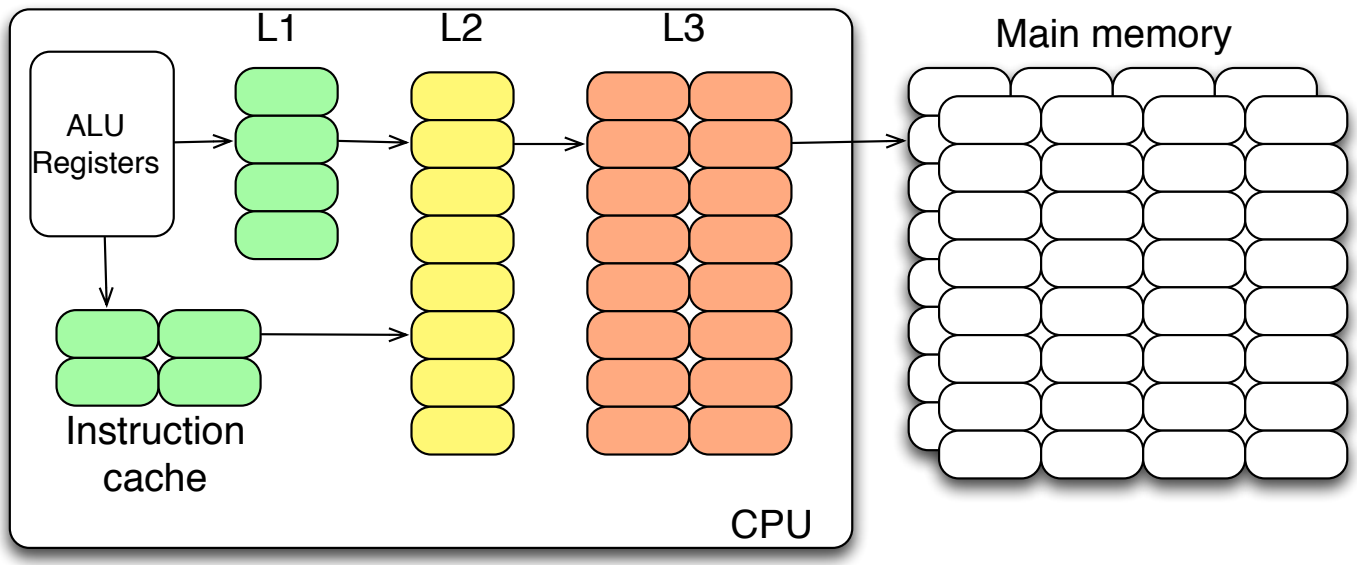


Instruction caches

So far we've mostly assumed that our caches can not only be read from, but also written to. This is vital for data.

But in most modern computers, instructions can only be read. Caches that are read-only are much easier technically and hence faster, and taking up less chip space.

Consequently, modern CPUs often have a separate and fast **instruction cache** that exploits instruction locality.



Interesting issues not mentioned

Interesting issues not mentioned

- Modern CPUs have multiple cores, i.e. mini-CPUs. How do cores and caches interact?

Interesting issues not mentioned

- Modern CPUs have multiple cores, i.e. mini-CPUs. How do cores and caches interact?
- Caches can be used to steal data from other processes (e.g. private keys), how can that be avoided? See e.g. the paper “CACHE MISSING FOR FUN AND PROFIT” by Colin Percival.

Interesting issues not mentioned

- Modern CPUs have multiple cores, i.e. mini-CPUs. How do cores and caches interact?
- Caches can be used to steal data from other processes (e.g. private keys), how can that be avoided? See e.g. the paper “CACHE MISSING FOR FUN AND PROFIT” by Colin Percival. Short summary: this is a serious problem, and Intel has changed it’s CPU architectures so that caches can be ’switched off’ when dealing with secret data.

Interesting issues not mentioned

- Modern CPUs have multiple cores, i.e. mini-CPUs. How do cores and caches interact?
- Caches can be used to steal data from other processes (e.g. private keys), how can that be avoided? See e.g. the paper “CACHE MISSING FOR FUN AND PROFIT” by Colin Percival. Short summary: this is a serious problem, and Intel has changed it’s CPU architectures so that caches can be ’switched off’ when dealing with secret data.
- Spectre, Meltdown...

Conclusion

Conclusion

Caches in modern CPUs are a form of fast memory that automatically exploits memory locality to speed up execution, often quite considerably.

Conclusion

Caches in modern CPUs are a form of fast memory that automatically exploits memory locality to speed up execution, often quite considerably.

Making good use of the cache typically has drastic influences on execution speed.

Conclusion

Caches in modern CPUs are a form of fast memory that automatically exploits memory locality to speed up execution, often quite considerably.

Making good use of the cache typically has drastic influences on execution speed.

It is currently difficult for compilers to increase data-locality (although Stop & Copy garbage collectors help).

Conclusion

Caches in modern CPUs are a form of fast memory that automatically exploits memory locality to speed up execution, often quite considerably.

Making good use of the cache typically has drastic influences on execution speed.

It is currently difficult for compilers to increase data-locality (although Stop & Copy garbage collectors help).

For most normal programming activities, it's probably not a good idea to worry much about data locality. It's probably more economical to think about better high-level algorithms (or use a faster computer) when you encounter performance problems.

Conclusion

Caches in modern CPUs are a form of fast memory that automatically exploits memory locality to speed up execution, often quite considerably.

Making good use of the cache typically has drastic influences on execution speed.

It is currently difficult for compilers to increase data-locality (although Stop & Copy garbage collectors help).

For most normal programming activities, it's probably not a good idea to worry much about data locality. It's probably more economical to think about better high-level algorithms (or use a faster computer) when you encounter performance problems.

For really high-performance computing, programming cache-aware is vital, and has a substantial (negative) influence on program structure and portability.