What Do Processors Do?

Let's start at the low level: computers do things.
What do they do?

Aside: there will be code examples in many languages in the middle of the course. You will know some of the languages, and not others. That's okay.

I'll do my best to explain the key points so you can figure out what's happening.

What Does A Programming Language Do?

By now, we have an implicit definition of a program like “a description of some work that a computer can do for us”.

A programming language lets a person create these descriptions. i.e. it's a way for a person to tell a computer what they want done.

What Does A Compiler Do?

Short and incomplete answer: turns your code into something that can run on a processor. Let's assume for now that it's machine code (or close enough: assembly code).

A programming language is designed to be some combination of (1) easy for a person to read/write, and (2) possible to translate to efficient machine code.

C is (said to be) a “low level” language: what the programmer expresses is very similar to what the processor is going to do.

Unfortunately, everybody is lying to you...

Let's look at some C:

```c
int a = 1;
int b = 2;
int c = a+b;
return c;
```

What we imagine happens (in pseudo-assembly):

```
mov a, 1          ; create variable a
mov b, 2          ; create variable b
add c, a, b       ; add into c
push c            ; put return value in place
ret               ; return
```

Those instructions are sent to the processor and execute in the order we wrote them. Obvious, but wrong.

What Do Processors Do?

Assertion: you can't really understand how your program runs without knowing how the processor runs it, at least at some level.

But, the way a computer is often described is from the 1970s. In particular:

- Sequential execution: one operation executes at a time. Once that is complete, the next one starts.
- Flat memory: accessing all parts of memory takes the same amount of time.

This is also the model of a processor captured in C: sequential execution and flat memory are underlying assumptions of the way the language looks.

Those statements haven't been true for consumer processors since the 1980s. (e.g. both false for i486 and ARM3, released in 1989.)

At best, the abstraction presented by C is very leaky. At worst it's very misleading.

I'm going to need a modern processor to talk about: an Intel Core i7 6700K, a 4-core Skylake architecture (6th gen Core).

Other modern desktop, laptop, mobile processors will be similar but I don't want to spend time on 10 examples.

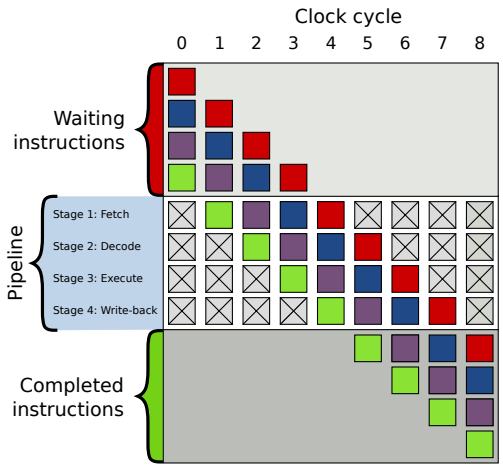
Sources for facts and/or further reading:

- [*] C Is Not a Low-level Language
- [†] Wikichip: Skylake microarchitecture
- [‡] Agner Fog's Optimization Manuals, specifically “for assembly programmers and compiler makers”.
- [‖] Wikipedia: Skylake
- [¶] i7-6700K product description
- Computer Systems: A Programmer's Perspective, Bryant & O'Hallaron. (i.e. the CMPT 295 text)

Pipelines

Modern processors are ***pipelined***: instructions are executed in several “stages”. Each instruction takes several processor cycles to execute.

e.g. a 4 stage pipeline would execute instructions like this: one instruction completes every cycle, but 4 are in motion at a time. *[Wikipedia: Pipeline]*



That means that for code like this...

```c
int a, b, c, d;
a = 2 - 1;
b = 1 + 1;
c = a + b;
d = 8 / 2;
```

... it seems like the values of `a` and `b` are **not yet computed** when the "`c =`" line starts to execute. How do we get the correct value in `c`?

Skylake: pipelines are 14–19 steps [†]. An instruction's result isn't going to be completely done for a *long* time.

Either the pipeline must ***stall*** (do nothing for a few cycles waiting for results) or maybe execute instructions out-of-order (e.g. move on to the "`d =`" line).

Wikipedia: Pipeline stall

Processors do both: programmers/compilers are not responsible for knowing the details of the pipeline.

If there are upcoming instructions independent of the ones in the pipeline, they will be started. If not, stall.

Skylake: can look up to 224 instructions away for something to execute out of order. [‡ §11.1]

It gets worse. Consider this code:

```c
int calculate_something(int a, int b) {
    int result;
    if ( a < b ) {
        result = a + b;
    } else {
        result = a - b;
    }
    return result;
}
```

As the condition (`<`) is evaluating, what should go into the pipeline next? An addition or a subtraction operation? Neither? Both?

Whenever there's a branch (i.e. a jump, goto, `if`, bottom-of-loop, etc.), the processor must do ***branch prediction*** and *guess* what's going to happen.

Code will be speculatively put into the pipeline. If the prediction was correct, then yay.

Wikipedia: Branch predictor

If the prediction was wrong, that execution must be discarded, the pipeline flushed, and the correct branch started. Penalty: number of cycles = pipeline depth.

If a branch at the bottom of a loop is mis-predicted often, the cost might be greater than everything happening in the loop.

Modern processors do lots of magic to avoid a mis-prediction.

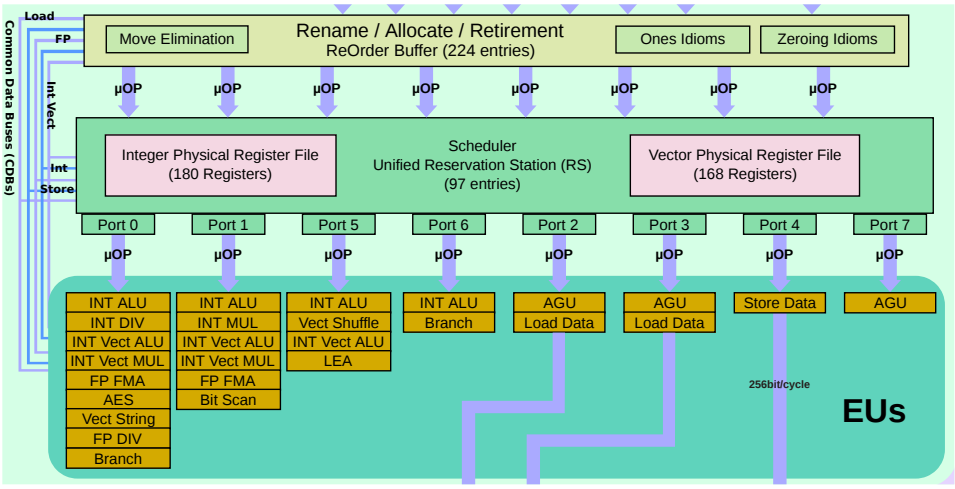
In some cases, a programmer can make their job easier with consistent patterns of branches. (e.g. It might actually be worth it to sort a list before processing, if that will make more consistent branches.)

Skylake: can detect patterns at least 29 branches long and predict that they'll continue. [‡ §3.8]

Modern processors also have more than one execution unit, so more than one pipeline that can be doing things concurrently.

This is where hyperthreading comes from: one processor core could try to run two separate threads to make the best use of its pipelines.

Skylake: there are 8 execution units that do specialized tasks. They could all be executing instructions all the time, but four is more realistic. [†, ‡ §11.9]



Multiple execution units mean *even more* instructions can be in-motion at once, making pipeline stalls or branch mispredictions even weirder.

Skylake: up to 180 instructions can be moving at one time [*].

SIMD

Another trick modern processors do: *SIMD instructions* (*Single Instruction, Multiple Data*). (You may know them by the Intel branding: MMX, SSE, AVX.)

The idea: you often want to do the same calculation on every element of an array. These instructions do the same operation on many values at once.

e.g. an array of many objects in 3D space that you want to project onto the 2D screen.

Skylake: supports AVX2 instructions, which have 256-bit wide SIMD paths. So, if you're operating on `double` values, you can work on 4 of them at a time; for 16-bit integers, 16 at a time.

If you use the serial-calculation instructions in the CPU, you'd do the same calculations 4–16 times slower.

SSE instruction list

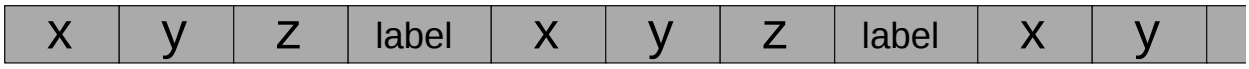
Good news: C compilers might notice that you have done SIMD-like things, and automatically transform your code for you: example of SSE instruction generation in C.

Bad news: only with `-O3` or `-ftree-vectorize`, or it might not figure out complex code.

The SIMD instructions rely on all of the values being *adjacent in memory*. If you have an array of `struct` or `class` values, then they won't be.

```c
struct scene_object {
    float x, y, z;
    char* label;
};
struct scene_object objs[10000];
```

An array of `struct` or `class` values is arranged like this in memory:



If you operate on every `objs[i].x` then they are aren't adjacent, so SIMD instructions can't be used. An array of `struct`/objects is something to be suspicious of if performance matters.

Multiple Cores

All processors you're likely to encounter have multiple *cores*. That is, multiple independant processing units which behave like processors on their own. (i.e. each has its own execution units, pipelines, SIMD instructions, etc.)

That includes mobile/embedded processors (often 4–8 cores, e.g. Snapdragon), laptop/desktop (often 4–16 cores), and server (often 4–32, e.g. Xeon, Epyc).

Our example Skylake: 4 cores.

Also remember hyperthreading: each core can run two threads simultaneously.

The implication: single-threaded code is **not** fast code.

From the programming language perspective, it would be very nice if the language made concurrent code easy and safe.

Memory Cache

By now, the “sequential execution” assumption is clearly false. Let's look at “flat memory”.

Your computer has some memory (RAM): maybe 8–32 GB of it. All of your variables live there. You can access it as you like (thus **Random Access** Memory). But...

Suppose we have a big array (that we think of as 2D)...

```c
const int size = 20000;
int *arr = (int*)malloc(size*size*sizeof(int));
```

... do something to initialize the values and sum...

```c
total = 0;
st = clock();
for(i=0; i<size; i++) {
    for(j=0; j<size; j++) {
        total += arr[i*size + j];      // sum by row
        // total += arr[j*size + i];   // sum by column
```

```
    }
}
en = clock();
```

Code as-is: 0.10 s. With the other array indexing: 1.3 s.

[Want try it? Download the code and compile with `gcc -O3`.]

Why does it matter so much (13× speed difference) whether we iterate by-row vs by-column?

Main memory is much slower than the processor: it takes dozens of cycles to fetch a value from RAM. We don't want the processor to wait that long.

Skylake: 4.00 GHz × 14.06 ns = 56 cycle memory latency [¶ with DDR4-2133].

To deal with this, processors have small amounts of faster memory that hold recently-used data from main memory: *memory cache*.

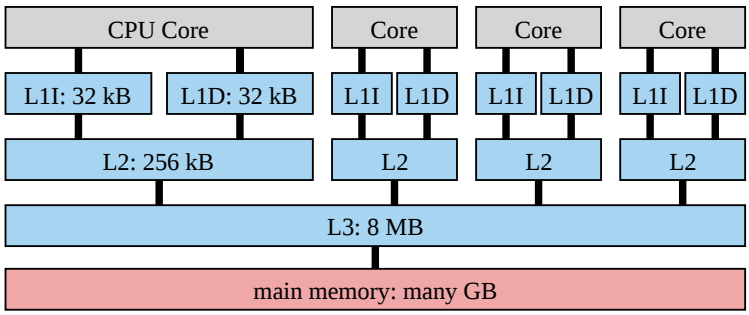
But capacity is limited: it will be holding a small amount of memory (a few pages) near what you have just accessed.

Implication: accessing memory close to what has been accessed recently will be many times faster than accessing something far away.

If you're going to traverse data, do it sequentially.

A (classic) linked list is an insane data structure to use in a modern memory architecture.

Modern CPUs have several levels of slower/larger/cheaper cache, possibly including separate cache for instructions and data. Our Skylake example: [†]



With multiple levels of cache, the penalty for accessing less-recently-used data might be less than a trip to memory, but still worse than hitting L1 cache.

The more local you can be in your memory accesses, the better.

One row in our 2D array example used 20000 `int`s × 4 bytes/`int` = 78 kB. Jumping to the next row was likely always an L1 cache miss.

The whole array was 1.5 GB, so we went to memory rarely when iterating across rows, and usually/often when iterating columns.

The lesson: it *really* matters how you access memory in your code.

If you're iterating through data, you need to know how it's arranged in memory.

Also, look back at the `struct` example: more unrelated data in one array will mean less cache locality.

Summary

The potential speedups over the dumbest single-threaded C code:

- × 2: good branch predictions (approx, depending on branch frequency),
- × 8: SIMD instructions (depending on data type),
- × 2: hyperthreading,
- × 4: multiple cores,
- × 10: hitting vs missing cache (approx, depending on miss frequency).

× 1280 between getting all of them wrong and right. That's a definite exaggeration for any real code, but knowing and using your processor is important.

Why is this in CMPT 383? Because programming languages can...

- let you write enough details to express what needs to be done;
- ... but hide details you don't care about (like order of iterating a many-dimensional array) behind higher-level descriptions;
- help you structure your data in memory well;
- help you write concurrent code that can use more cores.

Consider this equivalent C and Python code that adds one to each array element:

```c
int arr[1000];
// ... fill in array
for( int i=0; i<1000; i++ ) {
    arr[i] += 1;
}
```

.c

```python
import numpy
arr = numpy.empty(1000, dtype=numpy.int)
# ... fill in array
arr += 1
```

.py

The C code expresses exactly how to calculate, but not what's actually going to happen.

Some things the processor does are out of any programmer's hands, because they're complicated and incompatible between processors:

- pipeline details,
- branch prediction,
- when things move to/from cache.

But a programmer could write code that is better or worse for these.

Some things are under control of the programmer, and should be used effectively:

- SIMD instructions,
- multiple threads for multiple cores,
- how to structure data for good SIMD/cache performance.

Programming languages should help. Maybe C is not a low-level language.