We shall see that there are many kinds of parallelism
We shall see that there are many kinds of parallelism.

So we need to look at some of the ways hardware supports parallelism.
Hardware

Bit level

Recall from the 1st Year Architecture unit about adders: adding together two binary words
Hardware

Bit level

Recall from the 1st Year Architecture unit about adders: adding together two binary words

Serial Adders work one bit at a time, propagating the carry up the words as they do
Hardware

Bit level

Recall from the 1st Year Architecture unit about adders: adding together two binary words

Serial Adders work one bit at a time, propagating the carry up the words as they do

Simple hardware, simple to implement
Hardware

Bit level

Recall from the 1st Year Architecture unit about adders: adding together two binary words

Serial Adders work one bit at a time, propagating the carry up the words as they do

Simple hardware, simple to implement

Parallel Adders work on all the bits in parallel
Hardware

Bit level

Recall from the 1st Year Architecture unit about adders: adding together two binary words

Serial Adders work one bit at a time, propagating the carry up the words as they do

Simple hardware, simple to implement

Parallel Adders work on all the bits in parallel

More complex and expensive hardware, but faster
Hardware

Bit level

Recall from the 1st Year Architecture unit about adders: adding together two binary words

Serial Adders work one bit at a time, propagating the carry up the words as they do

Simple hardware, simple to implement

Parallel Adders work on all the bits in parallel

More complex and expensive hardware, but faster

A simple example, but this illustrates how parallelism trades complexity for speed
Pipelines

Again from Architecture: instructions are executed faster by using a pipeline
Pipelines

Again from Architecture: instructions are executed faster by using a pipeline

This is parallelism by overlapping the fetch → decode → fetch arguments → execute → store result cycle
Hardware

fetch → decode → args → exec → store → fetch → decode → args → exec → store → fetch → decode → args → exec → store...
Hardware

fetch → decode → args → exec → store → fetch → decode → args
→ exec → store → fetch → decode → args → exec → store → fetch
→ decode → args → exec → store...

becomes

fetch → decode → args → exec → store
  fetch → decode → args → exec → store
  fetch → decode → args → exec → store
  fetch → decode → args → exec → store
  ...

Again, more complexity for speed
It also shows how simple CPU clock speed is not a good indicator of speed of processing
A pipelined CPU will produce results faster than a non-pipelined CPU of the same clock speed
Hardware

fetch → decode → args → exec → store → fetch → decode → args → exec → store → fetch → decode → args → exec → store...

becomes

fetch → decode → args → exec → store
  fetch → decode → args → exec → store
    fetch → decode → args → exec → store
      fetch → decode → args → exec → store
        ...

Again, more complexity for speed
Hardware

fetch → decode → args → exec → store → fetch → decode → args → exec → store → fetch → decode → args → exec → store...

becomes

fetch → decode → args → exec → store
  fetch → decode → args → exec → store
    fetch → decode → args → exec → store
      fetch → decode → args → exec → store

Again, more complexity for speed

It also shows how simple CPU clock speed is not a good indicator of speed of processing
Hardware

fetch → decode → args → exec → store → fetch → decode → args → exec → store → fetch → decode → args → exec → store...

becomes

fetch → decode → args → exec → store
  fetch → decode → args → exec → store
    fetch → decode → args → exec → store
      fetch → decode → args → exec → store
        ...

Again, more complexity for speed

It also shows how simple CPU clock speed is not a good indicator of speed of processing

A pipelined CPU will produce results faster than a non-pipelined CPU of the same clock speed
Hardware

Coprocessors

Early chips were too small to fit everything on them
Coprocessors

Early chips were too small to fit everything on them.

So some operations were offloaded to a separate chip, a coprocessor.
Hardware

Coprocessors

Early chips were too small to fit everything on them

So some operations were offloaded to a separate chip, a *coprocessor*

At one point, a popular design was to put floating point operations on a coprocessor and only have integer arithmetic on the main processor chip
Hardware

Coprocessors

Early chips were too small to fit everything on them.
So some operations were offloaded to a separate chip, a coprocessor.

At one point, a popular design was to put floating point operations on a coprocessor and only have integer arithmetic on the main processor chip.

The coprocessor was specialised for floating point and could do little else.
Hardware

Coprocessors

Early chips were too small to fit everything on them

So some operations were offloaded to a separate chip, a coprocessor

At one point, a popular design was to put floating point operations on a coprocessor and only have integer arithmetic on the main processor chip

The coprocessor was specialised for floating point and could do little else

This allowed a weak form of parallelism: ship an operation (say a square root) off to the coprocessor, and while it is chewing on that, the main processor can carry on with something else in parallel
Coprocessors

Floating point eventually migrated onto the main chip (using lots of transistors!), but coprocessors are still hugely popular
Coprocessors

Floating point eventually migrated onto the main chip (using lots of transistors!), but coprocessors are still hugely popular.

Graphics cards (GPUs) are coprocessors, originally specialised to pixel crunching.

Exercise. Read about Tensor Processing Units (TPUs).
Coprocessors

Floating point eventually migrated onto the main chip (using lots of transistors!), but coprocessors are still hugely popular

Graphics cards (GPUs) are coprocessors, originally specialised to pixel crunching

And now they are commonly used as general purpose GPUs (GPGPU) and are turning out to be important in highly parallel computation
Hardware

Coprocessors

Floating point eventually migrated onto the main chip (using lots of transistors!), but coprocessors are still hugely popular

Graphics cards (GPUs) are coprocessors, originally specialised to pixel crunching

And now they are commonly used as general purpose GPUs (GPGPU) and are turning out to be important in highly parallel computation

We shall return to GPGPUs

Exercise. Read about Tensor Processing Units (TPUs)
Hardware

Coprocecssors

Floating point eventually migrated onto the main chip (using lots of transistors!), but coprocessors are still hugely popular

Graphics cards (GPUs) are coprocessors, originally specialised to pixel crunching

And now they are commonly used as general purpose GPUs (GPGPU) and are turning out to be important in highly parallel computation

We shall return to GPGPUs

Exercise. Read about Tensor Processing Units (TPUs)
Superscalar

To employ those extra transistors, engineers started putting multiple arithmetic units on the chip.
Superscalar

To employ those extra transistors, engineers started putting multiple arithmetic units on the chip.

For example, two add units.
Superscalar

To employ those extra transistors, engineers starting putting multiple arithmetic units on the chip.

For example, two add units.

The processor can now do two adds at the same time.
Superscalar

To employ those extra transistors, engineers started putting multiple arithmetic units on the chip.

For example, two add units.

The processor can now do two adds at the same time.

Simultaneous execution of whole instructions is called *superscalar*.
Superscalar

To employ those extra transistors, engineers starting putting multiple arithmetic units on the chip

For example, two add units

The processor can now do two adds at the same time

Simultaneous execution of whole instructions is called *superscalar*

Pipelining is parallel execution of *parts* of the instruction cycle
Hardware

For example, the two adds in

\[
\begin{align*}
x_1 &= y_1 + z_1; \\
x_2 &= y_2 + z_2;
\end{align*}
\]

can be done at the same time
Hardware

For example, the two adds in

\[ x_1 = y_1 + z_1; \]
\[ x_2 = y_2 + z_2; \]

can be done at the same time

However, the two adds in

\[ x_1 = y_1 + z_1; \]
\[ x_2 = x_1 + z_2; \]

cannot be done at the same time
Hardware

For example, the two adds in

\[ x_1 = y_1 + z_1; \]
\[ x_2 = y_2 + z_2; \]

can be done at the same time

However, the two adds in

\[ x_1 = y_1 + z_1; \]
\[ x_2 = x_1 + z_2; \]

cannot be done at the same time

The CPU needs to sort out the dependencies to determine if it can do simultaneous multiple operations
Hardware
Out of Order

This can be improved with careful *instruction scheduling* by the processor, to let it do *out of order execution*

For example, the code

\[
\begin{align*}
x_1 &= y_1 + z_1; \\
a_1 &= x_1 \times y_1; \\
x_2 &= y_2 + z_2;
\end{align*}
\]

is equivalent in results to

\[
\begin{align*}
x_1 &= y_1 + z_1; \\
x_2 &= y_2 + z_2; \\
a_1 &= x_1 \times y_1;
\end{align*}
\]

but on a CPU with two add units the latter can do the two adds in parallel
A processor that does out of order execution will scan the instruction stream, analyse the upcoming operations and their dependencies, and reorder them suitably.
A processor that does out of order execution will scan the instruction stream, analyse the upcoming operations and their dependencies, and reorder them suitably.

Implementing this in the hardware uses a lot of transistors, and so keeps the engineers happy.
Hardware
Out of Order

A processor that does out of order execution will scan the instruction stream, analyse the upcoming operations and their dependencies, and reorder them suitably.

Implementing this in the hardware uses a lot of transistors, and so keeps the engineers happy.

Compiler writers can help somewhat by generating machine code that is easier for the hardware to analyse.
A processor that does out of order execution will scan the instruction stream, analyse the upcoming operations and their dependencies, and reorder them suitably.

Implementing this in the hardware uses a lots of transistors, and so keeps the engineers happy.

Compiler writers can help somewhat by generating machine code that is easier for the hardware to analyse.

But, mostly, this is a hardware feature.
Hard Exercise (come back to this later). Suppose we have initial values $x = 0$ and $y = 1$. Two parallel threads on hardware that does out of order execution:

```
Thread 1    Thread 2
y = 3;      if (x == 1) {
   x = 1;      y = 2*y;
    }         }
```

What are the possible final values of $y$?

Example taken from the Rust website; also see https://en.wikipedia.org/wiki/Memory_ordering