Hyperthreading

The next stage is to duplicate the state-bearing parts of the processor, namely the Program Counter, the registers and other related stuff.
Hardware

Hyperthreading

The next stage is to duplicate the state-bearing parts of the processor, namely the Program Counter, the registers and other related stuff.

This allows two (generally two, sometimes more) simultaneous threads (streams of instructions) to share the available hardware.
Hyperthreading

The next stage is to duplicate the state-bearing parts of the processor, namely the Program Counter, the registers and other related stuff.

This allows two (generally two, sometimes more) simultaneous threads (streams of instructions) to share the available hardware.

There will be some conflicts between the threads if they both try to use a computational unit (say a division) when there is only one unit of that type on the chip.
Hardware

Hyperthreading

The next stage is to duplicate the state-bearing parts of the processor, namely the Program Counter, the registers and other related stuff.

This allows two (generally two, sometimes more) simultaneous threads (streams of instructions) to share the available hardware.

There will be some conflicts between the threads if they both try to use a computational unit (say a division) when there is only one unit of that type on the chip.

In that case one thread will have to pause and wait.
Hardware

The main argument for hyperthreading is that if one hyperthread has to wait for something (e.g., a memory access) the other can run and keep the core busy.
The main argument for hyperthreading is that if one hyperthread has to wait for something (e.g., a memory access) the other can run and keep the core busy.

The idea of having more threads of execution than hardware so that there is always a thread ready to run becomes very important later.
The main argument for hyperthreading is that if one hyperthread has to wait for something (e.g., a memory access) the other can run and keep the core busy.

The idea of having more threads of execution than hardware so that there is always a thread ready to run becomes very important later.

Hyperthreading gives the illusion of a multicore system, but is not truly multicore.
The main argument for hyperthreading is that if one hyperthread has to wait for something (e.g., a memory access) the other can run and keep the core busy.

The idea of having more threads of execution than hardware so that there is always a thread ready to run becomes very important later.

Hyperthreading gives the illusion of a multicore system, but is not truly multicore.

The amount of repetition in the architecture will imply some limits on how effective this is and how much parallelism can be gained, as will the pattern of memory accesses by the code.
Hardware

Some say that two hyperthreads are worth about 1.5 cores, due to the amount of interference between the threads.
Hardware

Some say that two hyperthreads are worth about 1.5 cores, due to the amount of interference between the threads.

Downsides are that the hyperthreads can fight over the core’s cache memory.
Some say that two hyperthreads are worth about 1.5 cores, due to the amount of interference between the threads.

Downsides are that the hyperthreads can fight over the core’s cache memory.

For some tasks hyperthreading can reduce overall performance.
Hardware

Some say that two hyperthreads are worth about 1.5 cores, due to the amount of interference between the threads.

Downsides are that the hyperthreads can fight over the core’s cache memory.

For some tasks hyperthreading can reduce overall performance.

And there are security issues where information can leak (via the cache) from one hyperthread to its pair.
Hardware

Some say that two hyperthreads are worth about 1.5 cores, due to the amount of interference between the threads.

Downsides are that the hyperthreads can fight over the core’s cache memory.

For some tasks hyperthreading can reduce overall performance.

And there are security issues where information can leak (via the cache) from one hyperthread to its pair.

Most High Performance systems turn off hyperthreading (more cache is more important than more threads).
Next: the idea of SIMD/vector processing has been adopted in a small way in the instruction sets of some processors
Next: the idea of SIMD/vector processing has been adopted in a small way in the instruction sets of some processors.

It arose from multimedia processing, graphics in particular.
Next: the idea of SIMD/vector processing has been adopted in a small way in the instruction sets of some processors. It arose from multimedia processing, graphics in particular. Some operations (e.g., computing pixel colours) are data parallel.
Next: the idea of SIMD/vector processing has been adopted in a small way in the instruction sets of some processors.

It arose from multimedia processing, graphics in particular.

Some operations (e.g., computing pixel colours) are data parallel.

Now we can regard a 64 bit register as:

- a 64 bit register
- two 32 bit registers
- four 16 bit registers
- eight 8 bit registers
An instruction is provided to (for example) add together eight 8 bit values in those registers in parallel
An instruction is provided to (for example) add together eight 8 bit values in those registers in parallel.

Another to add four 16 bit values in parallel, etc.
An instruction is provided to (for example) add together eight 8
bit values in those registers in parallel

Another to add four 16 bit values in parallel, etc.

\[
\begin{array}{c}
\text{+} \\
= \\
\end{array}
\quad \text{or} \quad
\begin{array}{c}
\text{+} \\
= \\
\end{array}
\]
Hardware

SWAR

This is *SIMD within a register* (SWAR)
Hardware

SWAR

This is *SIMD within a register* (SWAR)

We are treating the register as a (small) vector processor
This is *SIMD within a register* (SWAR)

We are treating the register as a (small) vector processor

This was found to be very effective for data parallel graphics processing
This is *SIMD within a register* (SWAR)

We are treating the register as a (small) vector processor

This was found to be very effective for data parallel graphics processing

Intel provide these instructions in their MMX (Multi Media Extensions), SSE (Streaming SIMD Extensions), SSE2, SSE3, SS4, AVX (Advanced Vector Extensions, 128 bit registers), AVX2 (256 bit registers) extensions
This is *SIMD within a register* (SWAR)

We are treating the register as a (small) vector processor

This was found to be very effective for data parallel graphics processing

Intel provide these instructions in their MMX (Multi Media Extensions), SSE (Streaming SIMD Extensions), SSE2, SSE3, SS4, AVX (Advanced Vector Extensions, 128 bit registers), AVX2 (256 bit registers) extensions

Similarly others from other manufacturers (AMD, Arm, etc.)
This is *SIMD within a register* (SWAR)

We are treating the register as a (small) vector processor

This was found to be very effective for data parallel graphics processing

Intel provide these instructions in their MMX (Multi Media Extensions), SSE (Streaming SIMD Extensions), SSE2, SSE3, SS4, AVX (Advanced Vector Extensions, 128 bit registers), AVX2 (256 bit registers) extensions

Similarly others from other manufacturers (AMD, Arm, etc.)

The process of converting sequential operations to SWAR is called *vectorisation*
Nice, but it needs compiler support to generate these instructions: rather than eight instructions to add eight 8-bit numbers, one instruction to add them in SWAR.
Nice, but it needs compiler support to generate these instructions: rather than eight instructions to add eight 8-bit numbers, one instruction to add them in SWAR

Plus a bunch of other stuff to get the values in and out of the right places in the register
Nice, but it needs compiler support to generate these instructions: rather than eight instructions to add eight 8-bit numbers, one instruction to add them in SWAR

Plus a bunch of other stuff to get the values in and out of the right places in the register

Compilers have always been far behind hardware: an architecture might provide an eight-way multiply instruction, but that is *only useful if you can get a compiler to generate code to use it*
Hardware

SWAR

Nice, but it needs compiler support to generate these instructions: rather than eight instructions to add eight 8-bit numbers, one instruction to add them in SWAR.

Plus a bunch of other stuff to get the values in and out of the right places in the register.

Compilers have always been far behind hardware: an architecture might provide an eight-way multiply instruction, but that is only useful if you can get a compiler to generate code to use it.

Or get the programmer to write the assembler by hand.
Nice, but it needs compiler support to generate these instructions: rather than eight instructions to add eight 8-bit numbers, one instruction to add them in SWAR

Plus a bunch of other stuff to get the values in and out of the right places in the register

Compilers have always been far behind hardware: an architecture might provide an eight-way multiply instruction, but that is only useful if you can get a compiler to generate code to use it

Or get the programmer to write the assembler by hand

For a compiler spotting that a loop can be converted into SWAR vector instructions is very hard
For example, the multiplies in the code

```c
char x[20], y[20];
for (i = 0; i < 20; i++) {
    y[i] = x[i]*x[i];
}
```

might be compiled as *three* \((8 + 8 + 4)\) 8-way SWAR multiply instructions
Making good compilers is harder than you think and has been a major drag on the effective use of modern hardware
Making good compilers is harder than you think and has been a major drag on the effective use of modern hardware.

A lot of code to use these kinds of instructions still has to be written by hand, in assembler.
In procedural code, we tend to write loops: the compiler would have to analyse it carefully to determine if SWAR would be useful.
In procedural code, we tend to write loops: the compiler would have to analyse it carefully to determine if SWAR would be useful.

In contrast, in the functional style we write code like “do this operation on these data” (map), which is much easier to analyse as the operation is explicitly separate from the iteration.
Exercise. Think about the code

```c
char x[], y[];
for (i = 0; i < n; i++) {
    y[i] = x[i]*x[i];
}
```

where the loop limit is variable

Exercise. Then think about the functional version

```java
y = x.map(square);
```
The transition of CPUs from *complex instruction set computer* (CISC) to *reduced instruction set computer* (RISC) architectures was based on advances in compiler technology.
The transition of CPUs from *complex instruction set computer* (CISC) to *reduced instruction set computer* (RISC) architectures was based on advances in compiler technology. The idea was to move complexity out of the hardware and into the software.
The transition of CPUs from *complex instruction set computer* (CISC) to *reduced instruction set computer* (RISC) architectures was based on advances in compiler technology. The idea was to move complexity out of the hardware and into the software. Rather than using complicated instructions poorly, we use simple instructions effectively.
The transition of CPUs from complex instruction set computer (CISC) to reduced instruction set computer (RISC) architectures was based on advances in compiler technology.

The idea was to move complexity out of the hardware and into the software.

Rather than using complicated instructions poorly, we use simple instructions effectively.

This is strongly reliant on the compiler being good enough to understand and exploit the details of the RISC architecture.
The transition of CPUs from *complex instruction set computer* (CISC) to *reduced instruction set computer* (RISC) architectures was based on advances in compiler technology.

The idea was to move complexity out of the hardware and into the software.

Rather than using complicated instructions poorly, we use simple instructions effectively.

This is strongly reliant on the compiler being good enough to understand and exploit the details of the RISC architecture.

But this is easier than a compiler trying to make best use of a complicated CISC architecture.
Hardware

VLIW

The same idea was touted for the *very long instruction word* (VLIW)
The same idea was touted for the very long instruction word (VLIW)

Design a processor with many repeated arithmetic units—lots of add units, lots of multiply units and so on
The same idea was touted for the *very long instruction word* (VLIW)

Design a processor with many repeated arithmetic units—lots of add units, lots of multiply units and so on

Have instructions that are *very long*, e.g., 128 bits or more
The same idea was touted for the *very long instruction word* (VLIW)

Design a processor with many repeated arithmetic units—lots of add units, lots of multiply units and so on

Have instructions that are *very long*, e.g., 128 bits or more

The instructions are composites of the simple operations, e.g., two adds, a subtract and a multiply could be bundled together in a single instruction
Hardware

VLIW

The compiler composes these instructions and makes sure there are no nasty interactions between the sub-instructions, e.g., none of the inputs to the sub-instructions are the outputs of any others of the sub-instructions
The compiler composes these instructions and makes sure there are no nasty interactions between the sub-instructions, e.g., none of the inputs to the sub-instructions are the outputs of any others of the sub-instructions.

The compiler does the hard work of sorting out interactions, leaving the hardware to blast on at full speed without checking or doing any reordering.
Hardware

VLIW

The compiler composes these instructions and makes sure there are no nasty interactions between the sub-instructions, e.g., none of the inputs to the sub-instructions are the outputs of any others of the sub-instructions.

The compiler does the hard work of sorting out interactions, leaving the hardware to blast on at full speed without checking or doing any reordering.

The compiler is promising to the hardware that nothing bad is going to happen if the hardware blindly executes the instructions as given.
Moreover, the chip uses less energy as it does not have the silicon to do instruction dependency analysis and reordering and the like.
Moreover, the chip uses less energy as it does not have the silicon to do instruction dependency analysis and reordering and the like. The analysis and reordering was done by the compiler.
Moreover, the chip uses less energy as it does not have the silicon to do instruction dependency analysis and reordering and the like.

The analysis and reordering was done by the compiler.

This appeared in the Bulldog compiler (early 1980s) and the Multiflow computer (late 1980s).
Moreover, the chip uses less energy as it does not have the silicon to do instruction dependency analysis and reordering and the like.

The analysis and reordering was done by the compiler.

This appeared in the Bulldog compiler (early 1980s) and the Multiflow computer (late 1980s).

It didn’t turn out to be terribly practical or popular.
Moreover, the chip uses less energy as it does not have the silicon to do instruction dependency analysis and reordering and the like.

The analysis and reordering was done by the compiler.

This appeared in the Bulldog compiler (early 1980s) and the Multiflow computer (late 1980s).

It didn’t turn out to be terribly practical or popular.

Compilers were not sufficiently clever to untangle enough instruction dependencies to get good hardware utilisation.
VLIW was briefly revived by Intel in their Itanium processor (2001)
VLIW was briefly revived by Intel in their *Itanium* processor (2001)

They called it *Explicitly Parallel Instruction Computing* (EPIC), a limited form of VLIW
VLIW was briefly revived by Intel in their *Itanium* processor (2001)

They called it *Explicitly Parallel Instruction Computing* (EPIC), a limited form of VLIW

It, too has flopped
VLIW was briefly revived by Intel in their *Itanium* processor (2001)

They called it *Explicitly Parallel Instruction Computing* (EPIC), a limited form of VLIW

It, too has flopped

Possibly due to their classic x86 chips being too entrenched, but also their compiler was never quite up to the job
It still pops up here and there: some AMD Radeon graphics chips have a VLIW architecture, though their newer architectures reverted to more traditional RISC
It still pops up here and there: some AMD Radeon graphics chips have a VLIW architecture, though their newer architectures reverted to more traditional RISC.

VLIW may well re-emerge in the future when compilers have progressed further: though more likely it will be overtaken by other kinds of hardware parallelism.
Exercise. Think about the example with VLIW

```c
char x[], y[];
for (i = 0; i < n; i++) {
    y[i] = x[i]*x[i];
}
```
Next we have full replication of arithmetic units, control and registers: true multicore
Next we have full replication of arithmetic units, control and registers: true multicore

Two or more full CPUs on the same chip
Next we have full replication of arithmetic units, control and registers: true multicore

Two or more full CPUs on the same chip

Often regarded as the first emergence of hardware parallelism
Next we have full replication of arithmetic units, control and registers: true multicore

Two or more full CPUs on the same chip

Often regarded as the first emergence of hardware parallelism

But, as we have seen, it’s not
Early multiprocessor machines were unicore chips side by side on the same motherboard.
Early multiprocessor machines were unicore chips side by side on the same motherboard.

Modern multicore processors, being on the same chip, can share things like on-chip cache memory and other chip infrastructure.
Early multiprocessor machines were unicore chips side by side on the same motherboard.

Modern multicore processors, being on the same chip, can share things like on-chip cache memory and other chip infrastructure.

Also there is faster inter-processor data transfer: no need to go off-chip. Off-chip transfers run at the bus speed, much slower than the chip speed.
Large machines tend to be multiple multicores: e.g., Balena has two 8-core chips on a motherboard; a total of 16 threads of execution (or 32 if the 2-way hyperthreading was enabled)
Large machines tend to be multiple multicores: e.g., Balena has two 8-core chips on a motherboard; a total of 16 threads of execution (or 32 if the 2-way hyperthreading was enabled)

This is slightly *asymmetric*: some processors are a little “closer” to each other than the others
Hardware

All of the above

These things are not mutually exclusive
Hardware

All of the above

These things are not mutually exclusive

A typical large installation these days is a CLUMP
Hardware

All of the above

These things are not mutually exclusive

A typical large installation these days is a CLUMP

- a cluster
Hardware

All of the above

These things are not mutually exclusive

A typical large installation these days is a CLUMP

- a cluster
- of multiple processors
Hardware

All of the above

These things are not mutually exclusive

A typical large installation these days is a CLUMP

- a cluster
- of multiple processors
- each having multiple cores
Hardware

All of the above

These things are not mutually exclusive

A typical large installation these days is a CLUMP

- a cluster
- of multiple processors
- each having multiple cores
- which might have hyperthreads
Hardware

All of the above

These things are not mutually exclusive

A typical large installation these days is a CLUMP

- a cluster
- of multiple processors
- each having multiple cores
- which might have hyperthreads
- and SWAR instructions
Hardware

All of the above

These things are not mutually exclusive

A typical large installation these days is a CLUMP

- a cluster
- of multiple processors
- each having multiple cores
- which might have hyperthreads
- and SWAR instructions
- on a pipelined architecture
Hardware

All of the above

These things are not mutually exclusive

A typical large installation these days is a CLUMP

• a cluster
• of multiple processors
• each having multiple cores
• which might have hyperthreads
• and SWAR instructions
• on a pipelined architecture
• with parallel instructions
Hardware

All of the above

These things are not mutually exclusive

A typical large installation these days is a CLUMP

- a cluster
- of multiple processors
- each having multiple cores
- which might have hyperthreads
- and SWAR instructions
- on a pipelined architecture
- with parallel instructions
- sometimes with a coprocessor or two on the side
Hardware

All of the above

These things are not mutually exclusive

A typical large installation these days is a CLUMP

- a cluster
- of multiple processors
- each having multiple cores
- which might have hyperthreads
- and SWAR instructions
- on a pipelined architecture
- with parallel instructions
- sometimes with a coprocessor or two on the side

It is very hard to make efficient use of all that!