Topics

We now look at a few topics in parallel computing

Topics

We now look at a few topics in parallel computing

Each year this unit is given may cover different topics so don’t be too worried if past exam papers ask questions on things that were not covered this year

Hardware

We have seen that there are many kinds of parallelism

Hardware

We have seen that there are many kinds of parallelism

But there has been hardware support for parallelism for much longer than you might think

Hardware

We have seen that there are many kinds of parallelism

But there has been hardware support for parallelism for much longer than you might think

Even in sequential CPUs!

Hardware

Bit level

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

Hardware

Pipelines

Hardware

Pipelines

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

Hardware

Pipelines

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

This is parallelism by overlapping the fetch$\rightarrow$decode$\rightarrow$fetch arguments$\rightarrow$execute$\rightarrow$store result cycle

Hardware

fetch→decode→args→exec→store→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
                                          …
```

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

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

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

Coprocessors

Hardware

Coprocessors

Early chips were too small to fit everything on them

Hardware

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

Hardware

Coprocessors

Hardware

Coprocessors

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

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

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

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

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

Superscalar

Hardware

Superscalar

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

Hardware

Superscalar

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

For example, two add units

Hardware

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

Hardware

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*

Hardware

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

```
x1 = y1 + z1;
x2 = y2 + z2;
```

can be done at the same time

Hardware

For example, the two adds in

```
x1 = y1 + z1;
x2 = y2 + z2;
```

can be done at the same time

However, the two adds in

```
x1 = y1 + z1;
x2 = x1 + z2;
```

cannot be done at the same time

Hardware

For example, the two adds in

```
x1 = y1 + z1;
x2 = y2 + z2;
```

can be done at the same time

However, the two adds in

```
x1 = y1 + z1;
x2 = x1 + z2;
```

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

```
x1 = y1 + z1;
a1 = x1*y1;
x2 = y2 + z2;
```

is equivalent in results to

```
x1 = y1 + z1;
x2 = y2 + z2;
a1 = x1*y1;
```

but on a CPU with two add units the latter can do the two adds in parallel

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

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 lots 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 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

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 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

Hardware

Out of Order

But we have already seen how out of order execution can break parallel code if we are not careful

Hardware

Out of Order

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`

Hardware

Hyperthreading

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

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

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

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

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 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

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 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

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

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 (a bigger share of the memory cache is more important than more threads)

Hardware

SWAR

Next: the idea of SIMD/vector processing has been adopted in a small way in the instruction sets of some processors

Hardware

SWAR

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

Hardware

SWAR

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

Hardware

SWAR

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

Hardware

SWAR

An instruction is provided to (for example) add together eight 8 bit values in those registers in parallel

Hardware

SWAR

An instruction is provided to (for example) add together eight 8 bit values in those registers in parallel

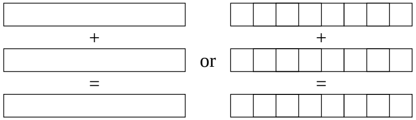
Another to add four 16 bit values in parallel, etc.

Hardware

SWAR

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.



SIMD Within A Word

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

Hardware

SWAR

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

Hardware

SWAR

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

Hardware

SWAR

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.)

Hardware

SWAR

Now, most code is written in a sequential fashion, e.g., looping over 8 values rather than code to add 8 values simultaneously

Hardware

SWAR

Now, most code is written in a sequential fashion, e.g., looping over 8 values rather than code to add 8 values simultaneously

In fact, few languages support SWAR operations directly, so there has to be some mechanism for getting to SWAR from conventional code

Hardware

SWAR

Now, most code is written in a sequential fashion, e.g., looping over 8 values rather than code to add 8 values simultaneously

In fact, few languages support SWAR operations directly, so there has to be some mechanism for getting to SWAR from conventional code

The process of converting sequential operations to SWAR is called *vectorisation*

Hardware

SWAR

We need compiler support to generate these SWAR instructions: it needs to spot that rather than generating eight instructions to add eight 8-bit numbers, it should generate one instruction to add them in SWAR

Hardware

SWAR

We need compiler support to generate these SWAR instructions: it needs to spot that rather than generating eight instructions to add eight 8-bit numbers, it should generate one instruction to add them in SWAR

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

We need compiler support to generate these SWAR instructions: it needs to spot that rather than generating eight instructions to add eight 8-bit numbers, it should generate one instruction to add them in SWAR

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 writer the assembler by hand

Hardware

SWAR

We need compiler support to generate these SWAR instructions: it needs to spot that rather than generating eight instructions to add eight 8-bit numbers, it should generate one instruction to add them in SWAR

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 writer the assembler by hand

For a compiler spotting that a loop can be converted into SWAR vector instructions is very hard

Hardware
SWAR

For example, the multiplies in the code

```
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

Hardware
SWAR

For example, the multiplies in the code

```
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

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

Hardware

SWAR

Making good compilers is harder than you think and has been a major drag on the effective use of modern hardware

Hardware

SWAR

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

Hardware

SWAR

In procedural code, we tend to write loops: the compiler would have to analyse it carefully to determine if SWAR would be useful (e.g., no value depends on an earlier value in the loop)

Hardware

SWAR

In procedural code, we tend to write loops: the compiler would have to analyse it carefully to determine if SWAR would be useful (e.g., no value depends on an earlier value in the loop)

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

Hardware

SWAR

Exercise Think about the code

```
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

```
y = x.map(square);
```

Hardware

VLIW

The transition of CPUs from *complex instruction set computer* (CISC) to *reduced instruction set computer* (RISC) architectures was based on advances in compiler technology

Hardware

VLIW

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

Hardware

VLIW

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: by streamlining the instruction set we can run things faster

Hardware

VLIW

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: by streamlining the instruction set we can run things faster

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

Hardware

VLIW

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: by streamlining the instruction set we can run things faster

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)

Hardware

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

Hardware

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

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

Hardware

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

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

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

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

Hardware

VLIW

Moreover, the chip uses less energy as it does not have the silicon to do instruction dependency analysis and reordering and the like

Hardware

VLIW

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

Hardware

VLIW

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)

Hardware

VLIW

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

Hardware

VLIW

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

Hardware

VLIW

VLIW was briefly revived by Intel in their *Itanium* processor (2001)

Hardware
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

Hardware

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

Hardware

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

Possibly due to their classic x86 chips being too entrenched, but also their compiler was never quite up to the job

Hardware

VLIW

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

Hardware

VLIW

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

Hardware

VLIW

Exercise Think about the

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

example with VLIW

Hardware

Multicore

Next we have full replication of arithmetic units, control and registers: true multicore

Hardware

Multicore

Next we have full replication of arithmetic units, control and registers: true multicore

Two or more full CPUs on the same chip

Hardware

Multicore

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

Hardware

Multicore

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

Hardware

Early multiprocessor machines were unicore chips side by side on the same motherboard

Hardware

Early multiprocessor machines were unicore chips side by side on the same motherboard

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

Hardware

Early multiprocessor machines were unicore chips side by side on the same motherboard

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

Also there is faster inter-core data transfer: no need to go off-chip. Off-chip transfers run at the bus speed, much slower than the chip speed

Hardware

Multicore

Large machines tend to be multiple multicores: e.g., two 24-core chips on a motherboard; a total of 48 threads of execution

Hardware

Multicore

Large machines tend to be multiple multicores: e.g., two 24-core chips on a motherboard; a total of 48 threads of execution

Or 96 if 2-way hyperthreading is enabled

Hardware

Multicore

Large machines tend to be multiple multicores: e.g., two 24-core chips on a motherboard; a total of 48 threads of execution

Or 96 if 2-way hyperthreading is enabled

This is slightly *asymmetric*: some cores 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!