Specialised processing architectures

- To date you have experienced the principle of processor operation …
  - Fetch, Decode, Execute
  - Integer arithmetic, logical operations {ADD, SUB, MUL, AND, OR …}
- … and seen some of the ways of implementing these

These operations can provide a ‘General Purpose’ processor

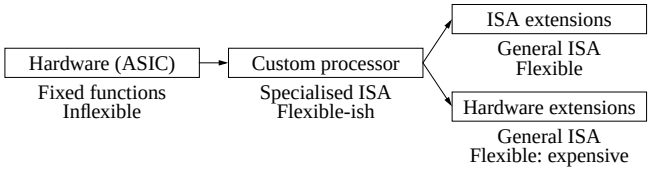
Specific applications demand particular other of processing: for example:

- Numeric processing – floating point arithmetic; vector processing
- Cryptography – ‘strange’ bit manipulation/transpositions; long word lengths
- Digital signal processing – real-time data streams; fast multiplication
- ‘Flashy’ graphics – very processor intensive but highly parallelisable
- …

This lecture also includes a short overview of floating point processing which is not-so-specialised-as-it-once-used-to-be.

History

Specialised processors appear where there is sufficient application demand and the needs cannot be filled by ‘general purpose’ devices. The evolution of the implementation of such applications tends to be:



- When something is feasible (and economic) a hardware solution may be built.
- As Moore’s Law provides more, faster resources more programmability is introduced
- As general purpose processors become more capable they can assimilate these functions
  - With the addition of extra instructions
  - With additional hardware
    - manufacturer’s extension
    - coprocessor – possibly designed by ASIC house

Example: DSPs

Specialised DSPs emerged around 1980 (cf. general purpose microprocessors from early 1970s). Early unit – a classic example was the TMS32010 – were very poor at ‘general’ code but provided high performance over a range of DSP applications. Early units were integer/fixed point; later floating point processing was incorporated. Later examples became closer to ‘general’ microprocessors whilst retaining DSP features (see later). ‘Dedicated’ DSPs still exist and are capable of more MIPS than general purpose cores – for a limited range of applications.

The ‘explosion’ of mobile (radio connected) computers probably helped sustain DSP development.

Often DSPs are used in conjunction with a ‘host’ processor in a heterogeneous multiprocessor system.

Programming

Programming is ‘specialised’ architectures is less amenable to ‘traditional’ languages than that for ‘general purpose’ microprocessors. This can be due to irregular instruction sets (such as many DSPs) or high degrees of parallelism (e.g. GPUs), often without explicit thread synchronisation. Similar issues can apply to the specialised ‘extensions’ to general purpose ISAs.

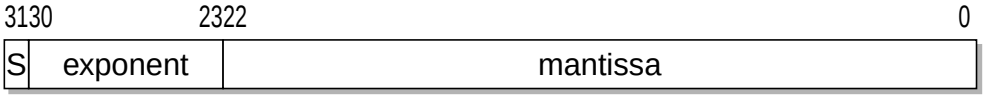
Programming these units often comes down to the use of hand-written (and optimised) libraries which provide an Application Programming Interface (API) to integrate them with ‘conventional’ code.

Examples, from the graphics world, could be OpenGL or DirectX.

Floating point representation

Many processors support floating point arithmetic.

- A floating point value is represented by a word-load of bits (shock!)
- The word is split into fields which represent different components of the value
  - analogy: the word "1234" could be interpreted as 123×10^4 … or 12×10^{34} … or …
  - decide in advance how the bits are to be interpreted!
  - standards exist – e.g. IEEE754 (32-bit format)



For most values this is interpreted as follows:

- The Sign bit indicates the sign of the value – '0' is positive, '1' is negative
- The exponent indicates the range of the number as a power of 2
  - the exponent is in excess form, a bit like a two's complement notation but: $7F_{16}$ (127_{10}) represents 0, 80_{16} (128_{10}) represents 1, $7E_{16}$ (126_{10}) represents -1 …
- The mantissa represents 1.<mantissa> as an *unsigned* value

Floating point representation

Normalised form

Illustrating with a decimal example, it is possible to represent the same value in different ways, e.g.

$1234 \times 10^0 = 123.4 \times 10^1 = 12.34 \times 10^2 = 1.234 \times 10^3 = 0.1234 \times 10^4$

A similar issue applies when representing in binary. and we can standardise on

This is redundant and we can standardise on and chosen form. 'Scientific' notation places the decimal point after the first non-zero digit.

In binary we can do the same thing.

The first non-zero binary digit *must* be a '1', so there's no need to store it! The mantissa is therefore *implicitly* 24 bits long.

Denormalised form

Normalised form prevents the storage of one very important value, i.e. '0'.

The minimum exponent value (i.e. 00) is therefore used to indicate that a value is denormalised.

These values are interpreted as $0.\text{<mantissa>} \times 2^{126}$

Thus, values which are smaller than the normalised minimum (2^{-126}) can be represented, albeit with less precision.

'Not a Number' (NaN)

The maximum exponent value (FF) is another special signal and is used to indicate values not otherwise representable. These include (both positive and negative) infinity and more bizarre values such as the result of 0/0.

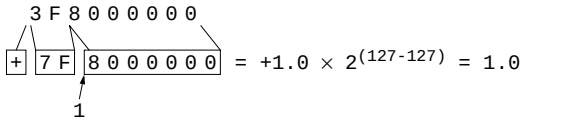
There are IEEE754 definitions for 16-bit ("half precision"), 32-bit ("single precision"), 64-bit ("double precision") and 128-bit ("quadruple precision") formats: in each case the *principle* of representation is the same.

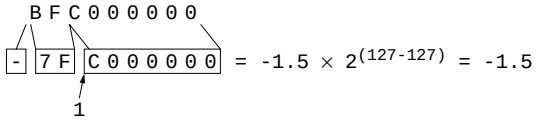
Examples

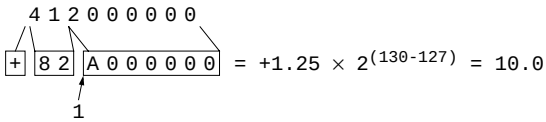
The slide shows a 32-bit floating point representation. IEEE 754 (as of 2008) defines four binary floating point representations.

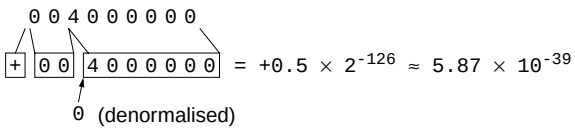
Name	Size (bits)	Mantissa (bits)	Exponent (bits)	Decimal digits	Minimum (approx.)	Maximum (approx.)
Half precision	16	10	5	~3	6×10^{-8}	65504
Single precision	32	23	8	~7	1.4×10^{-45}	3.4×10^{38}
Double precision	64	52	11	~16	4.9×10^{-324}	1.8×10^{308}
Quadruple precision	128	112	15	~34	6.4×10^{-4966}	1.2×10^{4932}

The following are single precision examples:









Floating point arithmetic

Multiplication

$(1.23 \times 10^4) \times (5.67 \times 10^8) = (1.23 \times 5.67) \times 10^{(4+8)} \approx 6.97 \times 10^{12}$ Note approximation!

- Mantissas are multiplied – easy because they are unsigned but usually need rounding
- Exponents are added – beware of range over- or under-flows
- Sign bits are XORed

$(2.13 \times 10^4) \times (5.67 \times 10^8) = (2.13 \times 5.67) \times 10^{(4+8)} \approx 12.08 \times 10^{12} \approx 1.21 \times 10^{13}$ Renormalising

- Note extra step may be needed to renormalise/round the result

Division is similar.

Addition and subtraction

$(9.87 \times 10^4) + (9.94 \times 10^6) = (0.0987 \times 10^6) + (9.94 \times 10^6) \approx 10.04 \times 10^6 \approx 1.00 \times 10^7$

- Mantissa of smaller value is **shifted** right until exponents are equal
- Mantissas are added (or subtracted) according to operation and sign bits
- Result may need renormalising after operation (particularly after a subtraction)

$(1.23 \times 10^4) - (1.20 \times 10^4) = 0.01 \times 10^4 = 1.00 \times 10^2$

Floating Point Processing

All operations first divide the word up into its various fields; for normalised values the implicit leading '1' is restored. Some floating point units will expand the fields into a larger representation internally to assist with rounding but also because it makes the implementation easier if (e.g.) both single and double precision values are passed through the same functional unit.

For multiplication (and division) the mantissas – which are just magnitudes – are multiplied (or divided). The sign bits are XORed, so if both the input numbers have the same sign the result is positive, if not the result will be negative. Multiplication will tend to produce more bits precision than the original values and may need rounding. division can iterate until the required precision is achieved. The exponents are added (or subtracted) – checking that the representable limits are not passed. The result *may* then need to be renormalised.

Addition (and subtraction) are actually more complicated to control. First the exponents must be compared and, if they are different, the smaller value's mantissa must be divided (shifted) until they are properly aligned and have a common exponent. After this the operation takes place: note that the mantissas are unsigned values so their sign bits, as well as the function must be considered in performing the operation. The result may or may not be normalised; if not it may need to be right shifted one place (if an add has 'overflowed') with a consequent increment to the exponent, or shifted left one or more places (if a subtraction yields a small (but non-zero) result) whilst decrementing the exponent appropriately. In all cases the exponent must be kept within its legal bounds or a zero or infinity may need to be signalled.

A floating point ALU is therefore somewhat more complex than an integer unit and needs some FSM control and sequencing.

Rounding

Floating point representations are, typically, *inexact*. Rounding is used to transform an answer into a valid representation. Different rounding modes are possible and results are defined in standards like IEEE754. For example 'to nearest' or 'towards zero'. Modern hardware needs to observe such standards too.

FPUs

Floating point operations are somewhat more complex and expensive to implement than integer operations. In early microprocessors there was no capacity for a floating point processing unit. When it was becoming apparent that hardware floating point operations would be desirable, the instructions started to be designed into the ISAs for implementation as a **coprocessor**. Later, as chip capacity increased, these were integrated with the integer processor and control unit but the design legacy remains.

A typical floating point unit is a computer datapath in itself, with registers and an ALU; it may have some of its own control logic too, for sequencing instructions, but it *does not fetch* its own instructions, it is slaved to a CPU.

A typical FPU will provide addition, subtraction, multiplication and division. Square root – or an iterative square root step – is also likely.

Transcendental functions (such as logs. and trig. functions) may still rely on software, using these basic algebraic operations.

Coprocessors

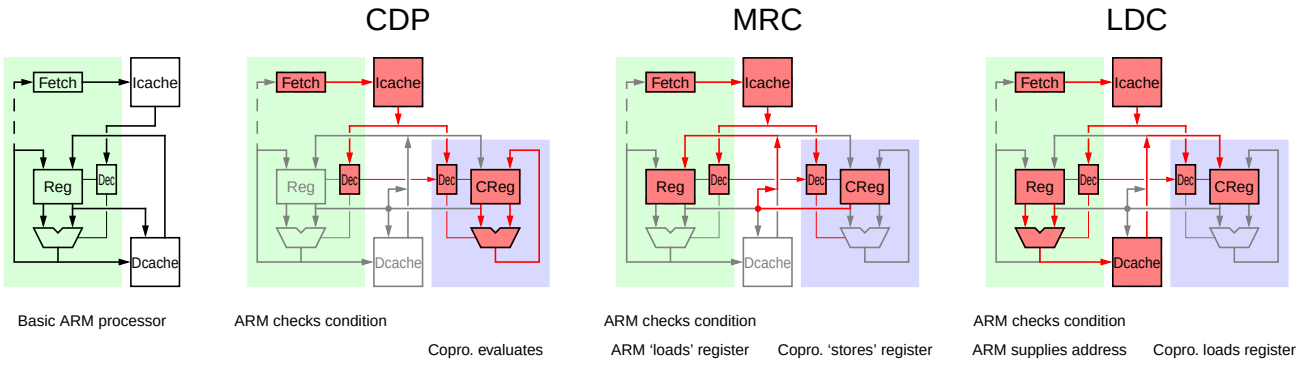
A coprocessor is a processing unit that is not (or, historically, was not) part of the basic ISA

It has (they have) some reserved space in the instruction set, used as needed.

A typical example would be floating point processing

Example: ARM has three classes of coprocessor operations

- Coprocessor data processing (CDP)
- Coprocessor register transfer (MRC/MCR)
- Coprocessor load/store (LDC/STC)



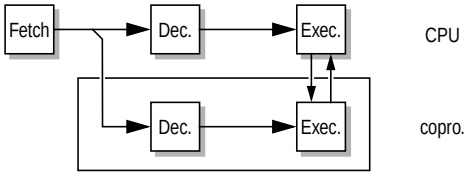
Operations *possible* using pre-existing bus structures.

ARM coprocessor operations

This column is here just to illustrate the types of operations that may be performed. The ARM is used as an example as it should be somewhat familiar.

ARM supports up to sixteen different coprocessors which are identified by a 4-bit field in the appropriate instructions.

A 'typical' ARM coprocessor may have sixteen internal registers plus a processing unit. It monitors the fetched instruction stream and ignores any operations it doesn't recognise. If it finds an instruction it can execute it will do so, in correspondence with the CPU.



Note that ARM has *conditional* coprocessor operations and the conditions are evaluated in the CPU, so an instruction may not complete, even if it *could*.

The classes of operations are:

- Data processing: the coprocessor performs an internal operation (such as a floating point multiplication
- Register transfer: a value is moved to an ARM register from a coprocessor register, or vice versa
- Load/Store: a coprocessor register value is moved from or to memory; the CPU calculates and supplies the address

If a coprocessor instruction is not taken by a coprocessor, the ARM treats it as an **illegal instruction** and an exception occurs; this allows the operating system to substitute a software model for the 'missing' hardware. If this is done it is called an '**emulator trap**' and it allows object code to be run even if all the appropriate hardware is not there. (It's much slower, of course!)

Example uses (in ARM)

One example use is a 'standard' processing unit such as ARM's own 'Vector FPU', available as Intellectual Property (IP) from ARM Ltd. This example implements all classes of coprocessor operations. Compilers will support the instructions and they have specific assembler mnemonics such as:

```
vmul.f64  d1, d2, d3; Double precision multiply
[cdp    p11, 2, c1, c2, c3, 0]
vmov    r2, r3, d4  ; Move double to ARM registers
[mrc    p10, 1, r2, c2, c0]
```

Another use would be for a hardware designer to add specific operations of their own to the instruction set. This may be useful if, for example, building a cryptographic processor which may rely on multiple bit transpositions: such operations are slow to perform in software.

Not all coprocessors need to support all forms of instructions. For example a common coprocessor on ARM systems is CP15 – the system control coprocessor. This does no 'processing' but acts as a set of hardware registers outside both the processor and the normal address space. It contains values such as the start address of the page tables, bits to enable caches and suchlike.

Example: coprocessor 'as was'

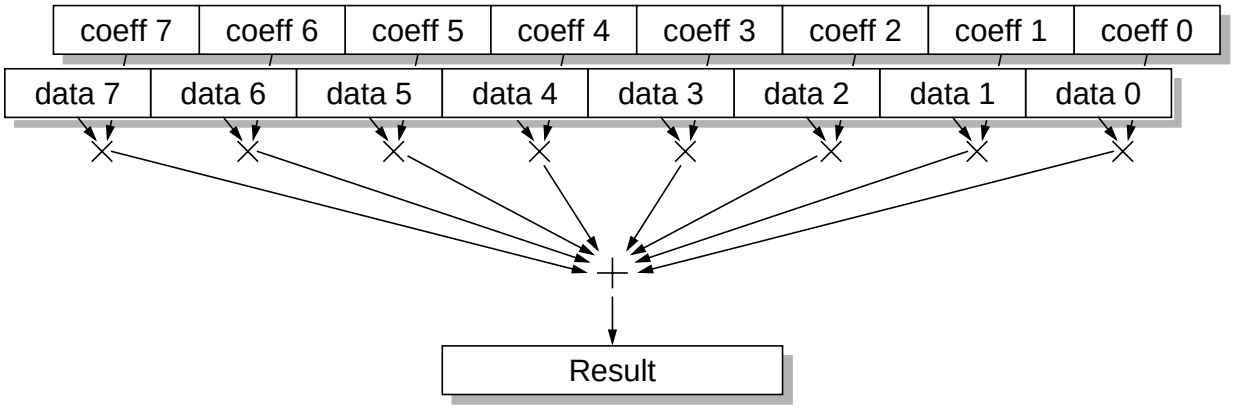
The 8086 (late 1970s) introduced (the first version of) the x86 ISA; it was an integer processor. The 8087 floating point coprocessor (1980) was a companion device. It came from different ancestry as can be seen by its internal structure which was a *stack-based instruction set*. Later x86 processors were also accompanied by FP coprocessors until they were integrated onto a single chip. The stack-based legacy can still be seen within the architecture.

The 8087 was influential in establishing what later became the IEEE754 floating point standard.

Digital Signal Processors (DSPs)

Specialised for high performance in a particular subset of computing tasks (such as filtering, FFT)

E.g. a digital filter requires **multiplying** two **vectors** and **accumulating** the result:




```
Result = 0;
for i = 0 to vector_length - 1
  Result = Result + data[i] * coeff[i];
```

The core of this will look something like:

```
repeat vector_length, #1,
MLA [a0] [a1]; INC a0; INC a1;
```

Where the second line is a single VLIW instruction and the loop count is done in hardware.

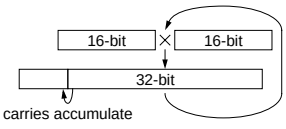
Typical DSP requirements

Typical DSP requirements	Typical DSP features
Streaming large data sets (vectors): Many data fetches	More than one independent memory bus: parallel access Often Harvard architecture
Often using *sampled* values	May only use 8- or 16-bit data elements
Many multiplications	Fast multiply-accumulate hardware
(Result may overflow)	Extra-long accumulator Clipping/saturating operations
Highly repetitive operations	Zero-overhead loops
'Unusual' addressing patterns	Special indexing patterns (for FFTs etc.)
Arithmetic-intensive processing	Explicit parallelism in instructions (VLIW) Automatic address indexing using parallel ALUs

Examples

- A 40-bit accumulator can allow 256 16×16 multiply-accumulates without overflowing.

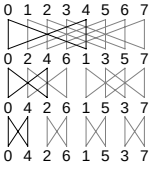
```
A := 0;
for i = 0 to 255
  A := A + x[i] * y[i];
```



- VLIW gives addressing parallelism

```
A := A + [i0] * [i1]; i0 := i0 + 1; i1 := i1 + 1;
```

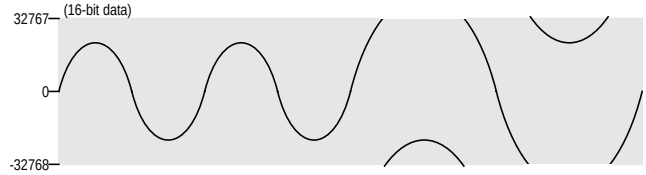
- Reordering address bits facilitates FFT address patterns



Indexing: pass 1		Indexing: pass 2		Indexing: pass 3		Indexing: final pass	
0	000	0	000	0	000	0	000
4	100	2	010	1	001	4	100
1	001	1	001	2	010	2	010
5	101	3	011	3	011	6	110
2	010	4	100	4	100	1	001
6	110	6	110	5	101	5	101
3	011	5	101	6	110	5	011
7	111	7	111	7	111	7	111

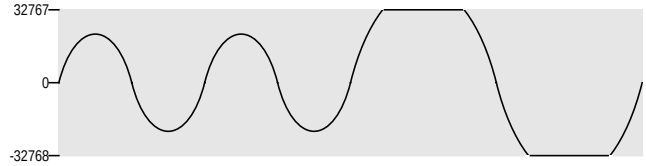
Saturating arithmetic

In two's complement arithmetic, adding two 'large' positive numbers may give a 'negative' result, if the available number range **overflows**.



Many processors will detect this and note the problem with an overflow flag (or similar). ARM and x86 both do this. This can be tested in software. Some languages choose not to (e.g. C, Java). This saves time but may give 'surprising' results.

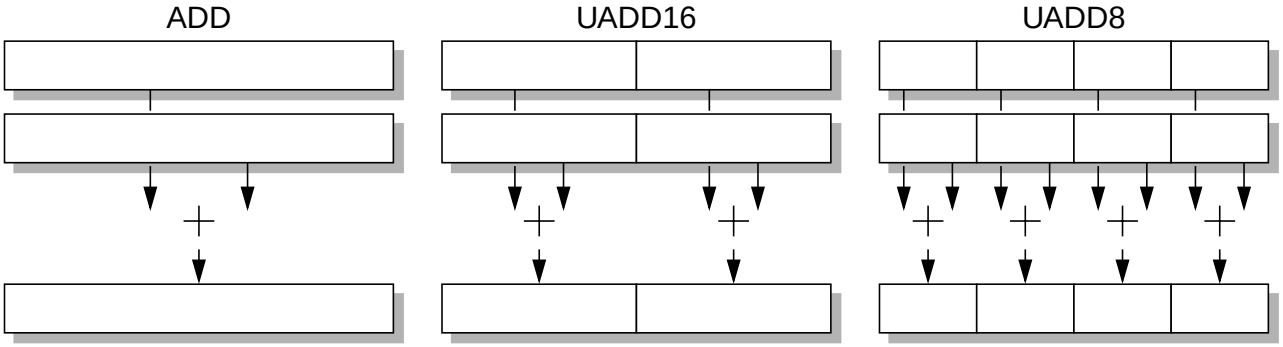
In DSP, sometimes it's better to simply 'clip' the value to the maximum representable one. This is called 'saturation'. A similar approach is taken to the maximum representable negative value.



Additional flags *may* be present to indicate that this has happened.

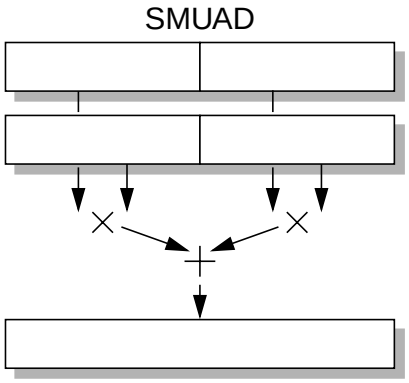
Superscalar extensions

Break up 32-bit registers into vectors, as specified by particular instructions:



These operations increase **stream** throughput on 'small' numbers using existing hardware (with quite minor changes).

E.g. in the ADDs (above) there are some gates preventing certain carries from propagating.



Superscalar extensions

'General purpose' processors continue to become more powerful and take over some of the 'specialist' roles of devices like DSPs. The demand for high-performance data streaming – first audio, then ever-increasing video resolution – puts a demand on workstation/mobile processors.

In response to this, both instructions and new hardware have been added to 'general purpose' processors to give additional support in areas such as DSP.

Intel introduced the 'MMX' extensions to the x86 ISA in 1997 and have subsequently developed this through multiple generations of Streaming SIMD[1] Extensions (SSE).

ARM has adopted a similar approach, both introducing (in ISA v.6) instructions which act on existing registers in the integer datapath and enhancements with extra functional units: the latter is called 'NEON'.

In all cases the *basic principle* is the same.

When processing audio data there is no need to use 32-bit data as the ear will not be sensitive to that resolution and the analogue electronics used to record/playback the sound is not precise to 1 part in billions!

Similarly, human eyes will not discriminate more than about 256 levels of brightness so 8 bits (arguably per colour) is sufficient.

Given a 32-bit ALU, half of more of the datapath can be wasted on any cycle. When data is *streamed* the same operations are performed on successive data.

It is therefore possible to pack more than one value into a word and process them in parallel, reducing the overall number of cycles. This is not trivially possible purely in software because values are independent so (for example) a carry output should not propagate from one value to the next in the register. **Saturation** of values may also be important in some applications.

The 'new' instructions do things like:

- four 8-bit operations at once
- two 16-bit operations at once
- two 16-bit multiplications then adding/subtracting the results
  - e.g. finding the modulus of a complex number
- Arbitrary size saturation

Vector Floating Point (VFP)

ARM's current[2] floating point accelerator is a 'vector' unit. It is a coprocessor – actually inhabiting *two* coprocessor spaces (CP10 & CP11) – and, depending on the version, adds 16 or 32 64-bit registers intended to hold double precision IEEE 754 variables. (Single precision values can be extended for processing.)

Vector processing

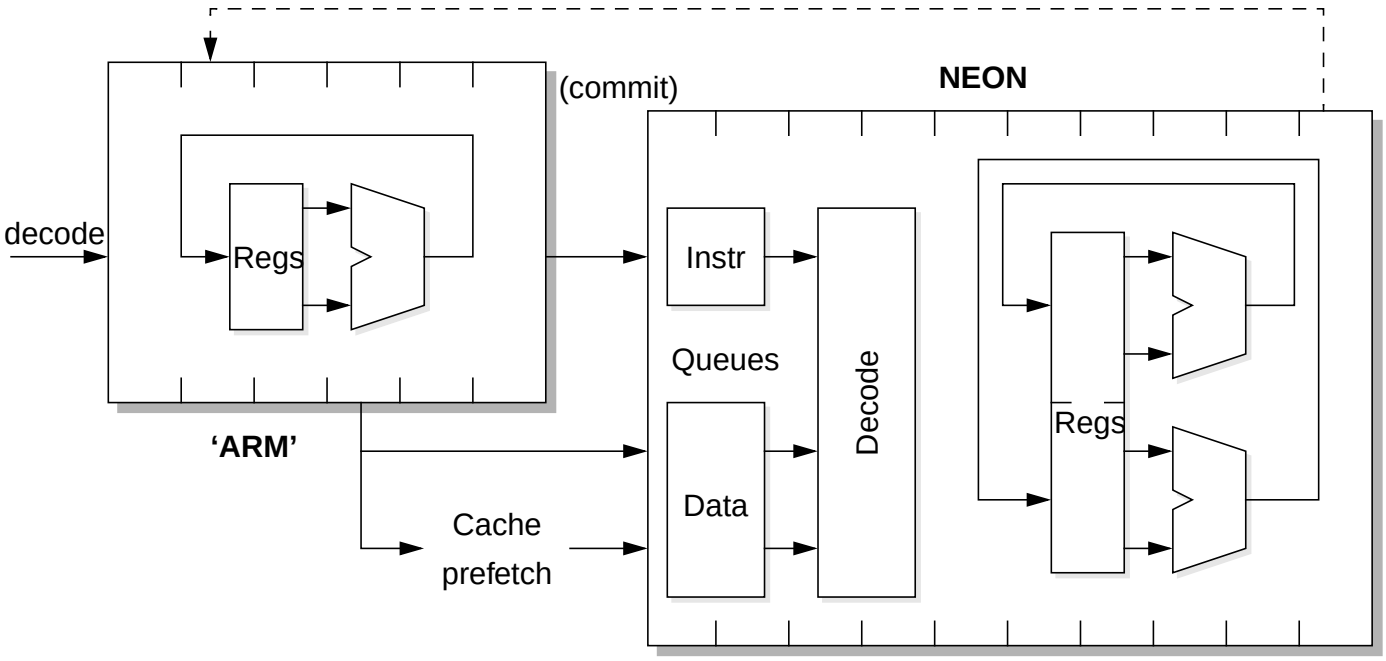
"Vector" is used here meaning a (one dimensional) array. 'Vector processing' is processing such data structures. This implicitly involves multiple similar calculations but it does not always mean that there is parallel processing. ARM's VFP processes vectors *serially*, internally, through a single functional unit. NEON (and the integer extensions) process several data in parallel.

Both these approaches may be called 'SIMD' and both are faster than coding operations in SISD software; clearly the latter approach is the most efficient though.

1. Single Instruction, Multiple Data: see Flynn's taxonomy (1966)

2. There has been at least one previous architecture, now obsolete.

NEON



- Committed NEON ops. are streamed out, queued and scheduled after the scalar stages.
- NEON processing stages comprise **many parallel pipelines** {IADD, IMUL, FADD, FMUL, …}

NEON

NEON

NEON is a superscalar extension processor which also appears as CP10/CP11. It uses 32 64-bit local registers (the same registers as the VFP, if implemented) which can be addressed as 16 128-bit registers. These are treated as vector registers; each can contain arrays of 8-, 16-, 32- or 64-bit integers or 32-bit floating point values.

- NEON is implemented as a deep (e.g. 10-stage) pipeline which succeeds the integer processing pipeline.
- The decision to complete an instruction is made in the integer pipe.
  - NEON instructions will complete sometime later
  - NEON *cannot generate exceptions*
- The integer pipe can initiate data fetches in time for input data to enter the top of the pipe
- Most processing happens 'later'; this is not usually a bottleneck because the NEON pipeline is separate.
- May be a long wait to get results back from NEON to 'main' registers
  - Hopefully, such transfers are not often needed.

Graphics Processing Units (GPUs)

Like DSP - special purpose requirements

- Latency tolerant
- Data parallel
- (Potentially) thousands of threads

- Streaming data
  - Not reused – no large caches
  - Can DMA in blocks from DRAM

- Can use 'synchronous' threads – no need for waiting and interlocking
  - Synchronisation handled at a higher level

- Much responsibility for correct function transferred to software; e.g. …
  - Memory (cache?) coherency
  - Thread synchronisation

GPUs

Graphics has 'traditionally' been one of the most compute-intensive applications.

- Large data sets
  - Over 1 million pixels is common
  - Significant memory (bandwidth) requirement
- (Often) real-time requirement
  - ~60 fps desirable, >15 fps may be necessary
- Multiple operations per pixel
  - '3D' objects may obscure each other
  - May want colour blending, etc.
- Intensive, repetitive calculations
- Highly parallelisable
  - Different sections of screen largely independent
- Latency tolerant
  - Providing *bandwidth* met, latency can be large

All this suggests that custom hardware acceleration is highly appropriate and has been employed for a long time. Given the hardware resource, more operations – such as object-to-screen transformations as well as drawing – can be accommodated.

From this point, a little more programmability can enable a number of other applications.

Programming

The 'usual' approach to programming GPUs is to use an abstracted API (Application Programming Interface) which allows portability to different architectures. *Object code compatibility* and *parallelisation* details are then not issues for most users. Probably the most common graphics API is **OpenGL** (Open Graphics Library) which (arguably) predates the introduction of GPUs.

Applications in 'general purpose' computing

With sufficient programmability it is possible to use GPUs for processing other massive data sets, where parallelism is appropriate and latency is not an issue. Their characteristics are such that they pack more functional units onto a silicon die than can be done with a 'true' general purpose processor. This is because processing can be maintained without the need for large caches.

General Purpose GPU (**GPGPU**) programming has gained a lot of recent interest and is employed in areas like bioinformatics, for instance searching DNA sequences. This is currently a developing field.

Some language 'platforms' have developed to support such programming e.g.:

- CUDA (Compute Unified Device Architecture) – NVidia
- DirectCompute – Microsoft
- Open Computing Language (OpenCL) – Khronos Group (open)

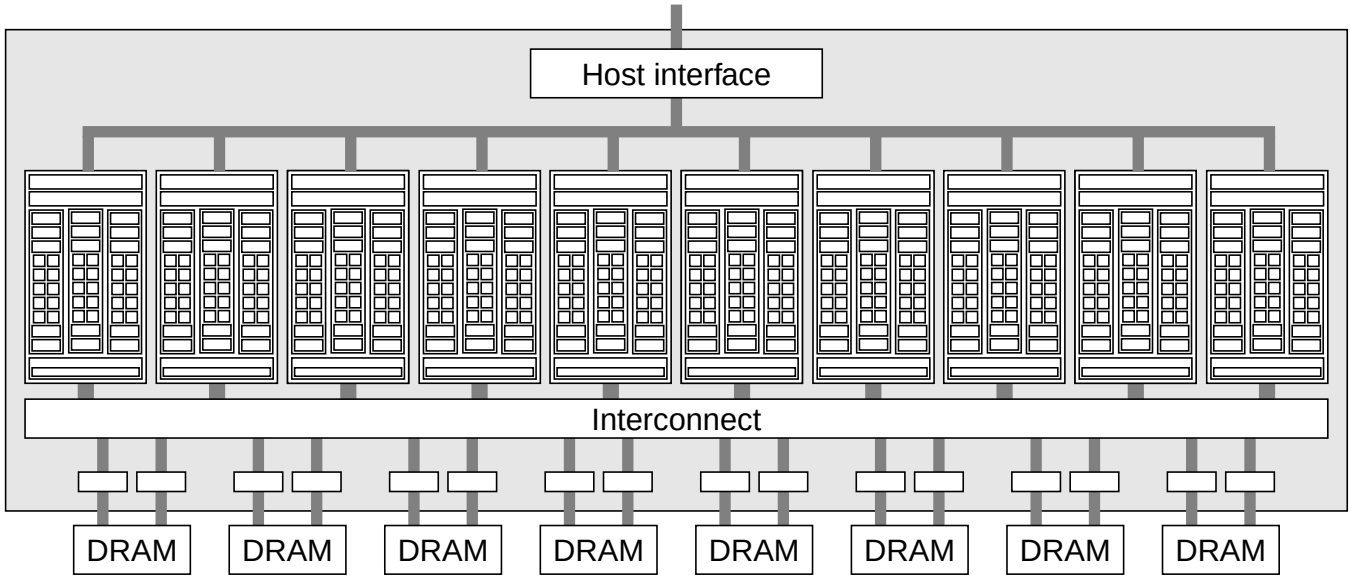
OpenCL

Open Computing Language (OpenCL) is a relatively recent (2011) attempt to provide a 'framework'[1] for programming heterogeneous systems. It is already quite usual to have multiple different types of processor on a SoC (System on Chip) and this is predicted to increase in the future. Therefore it is likely that future systems will contain specialised DSPs, GPUs (or a future evolution thereof) as well as general purpose processors. There will be architectural differences in these. Work will be apportioned to whichever systems are deemed appropriate at the time with unused processors 'sleeping' – or even turned off – to save power.

OpenCL has already been adopted by numerous major processor developers.

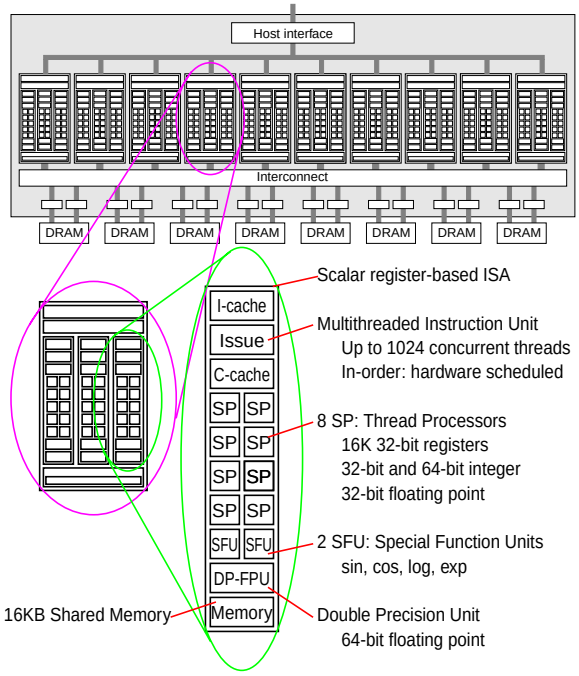
1. Bzzz … jargon alert!

Example GPU architecture (NVidia)



- Regular architecture: many blocks of processors ('Stream Multiprocessors')
- Big register sets – shared by many threads simultaneously
- (Quite) a lot of memory bandwidth using parallelism
- Switch threads whilst waiting for memory: don't bother cacheing

NVidia 'Stream Processor'



- Loosely taken from an amalgam of NVidia processors
- 'Other GPU manufacturers do exist'

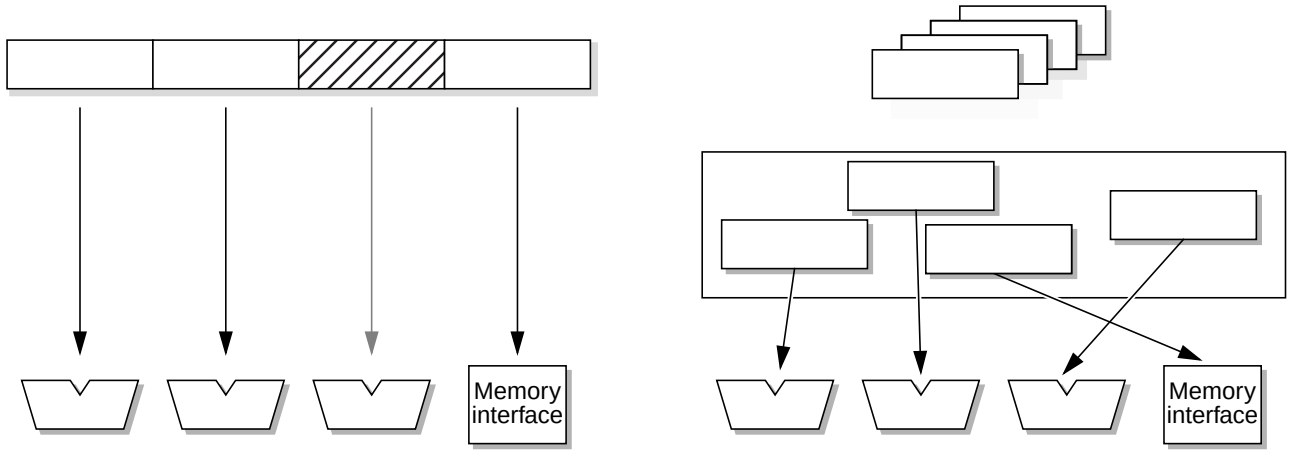
GPU features

- Large 'register' set in each processor
  - 'soft' partitioning into threads
- Relies on massive multithreading for performance
  - hardware scheduling – single cycle thread switch
  - limited/no thread interlocking
- "Blocks" of threads are partitioned amongst many processors
- (Data) memory latency high
  - *not cached* by hardware
  - concealed by thread switching instead
  - may have *tens of thousands* of threads per chip at any time

Multiprocessor Programming Model

- Workloads are partitioned into blocks of threads among multiprocessors
  - a block runs to completion
  - a block doesn't run until resources are available
- Allocation of hardware resources
  - shared memory is partitioned among blocks
  - registers are partitioned among threads
- Hardware thread scheduling
  - any thread not waiting for something can run
  - context switching is free – every cycle

Very Long Instruction Word (VLIW) & Superscalar



- Both issue (potentially) more than one instruction at once
  - VLIW schedules these with the compiler
  - Superscalar schedules at run time
- Both need wide instruction fetch buses to avoid starvation
  - Superscalar uses this more efficiently: no need for 'padding'
- Superscalar potentially faster but 'serious' hardware required ⇒ more area & power

VLIW

VLIW addresses the issue of superscalar instruction issue by making several instructions visible in a single instruction word. The instructions can be scheduled – and dependencies resolved – at compile time. This alleviates the need for the processor to schedule operations 'on the fly' which saves hardware, power etc.

VLIW has some disadvantages too.

- Static scheduling is not as efficient as dynamic scheduling
  - for example the optimal scheduling may vary on the first and subsequent iterations of a loop body
- Not infrequently there are not enough instructions ready for issue and some of the functional units cannot be used
  - this means inserting 'NOP's with a consequent waste of fetch bandwidth
- The format is tied to a limited number of implementation options (i.e. functional units) so *forwards compatibility* suffers.

VLIW tends not to be much used (currently) for general purpose computing. It is more common in (e.g.) dedicated DSPs where many functional units {Fetch unit, Multiplier, ALU, Address/Index units, …} tend to operate concurrently.

Superscalar Issue

Still needs considerable **fetch bandwidth** (which can be satisfied with a wide instruction bus) but instructions are delivered into a buffer from where they are issued. The 'next' instruction can be issued (unless there is a stall – e.g. waiting for a load) and subsequent instructions can go *unless there is a dependency* until all the functional units are busy. Although it does not have to be so, it would be normal to issue instructions out of order if they can start and there is processing capacity.

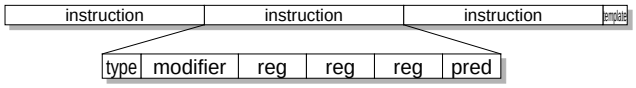
Superscalar issue is more efficient than VLIW, both in fetch bandwidth usage and in better (run time) scheduling but carries a cost in hardware complexity and power consumption. The details of issuing and reordering to give the appearance of sequential issue to the programmer exceed the bounds of this module!

EPIC/Itanium

Probably the best current general purpose example of a VLIW-ish architecture is Intel's Itanium (a.k.a. IA-64). This is not a mass-market architecture but is used for high-end servers and has been (and continues) in development since 2001.

Itanium is a 64-bit ISA with 64-bit registers and a 64-bit address space. It is not the same as the x86-64 used in PCs. There are 128 integer/pointer registers, a similar number of (82-bit) floating-point registers and, instead of flags, 64 1-bit predicate registers.

The instruction encoding philosophy is (rather catchily!) called "EPIC" (Explicitly Parallel Instruction Computing). Itanium instructions come in 128-bit words which contain three instructions and a 'template' field. The template specifies which operations can be executed in parallel, avoiding the need to pack in NOPs. The instructions are each 41 bits in length which allows space for (example) three register specifiers (@7 bits each) and a predicate specifier (@ 6 bits) as well as the op-code, etc.



Transport triggered architectures

All the systems described earlier in this lecture still use some conventional-looking microarchitectures. There is not room for much description here, but a 'transport triggered architecture' is a bit more different.

Instead of thinking of one (or more) datapaths which *contain* functional units, think of a bunch of functional units with addressable input *ports*; functional units could be 'add', 'sub', 'mul', registers etc.

Multiple 'move' operations are fetched (VLIW-style) by a control unit which then each route a function output to a function input. When a function unit has a set of inputs (e.g. two things to multiply) it can start operation. FIFOs may help smooth out operational timing.

The result is more of a mapping of a data-flow graph onto hardware than the 'conventional' fetch-decode-execute FSM.

A commercial example exists! Try looking up 'MAXQ'