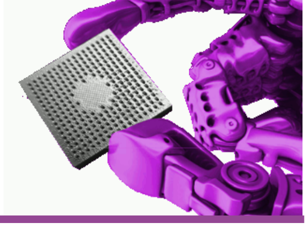
Computer Architecture

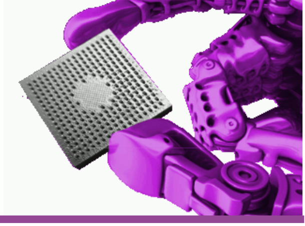
An embedded approach



Module 2

Laying the Foundations

Contents



2.1 Computer Organisation

2.2 Computer Fundamentals

2.3 Number Formats

2.4 Arithmetic

2.5 Multiplication

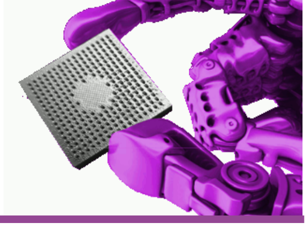
2.6 Division

2.7 Fractional Number Formats

2.8 Floating Point

2.9 Floating Point Processing

Computer Organisation



What is inside a computer?

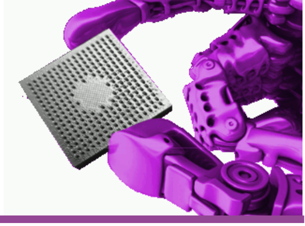
Blocks that move and manipulate binary data (traditionally, things like an arithmetic logic unit – ALU, memory, registers and so on).

What do these things do?

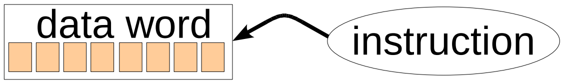
Nothing more than either *move* or *transform* binary values. All of our data (videos, pictures, mp3s etc.) are stored as large amounts of binary data...

How are these connected?

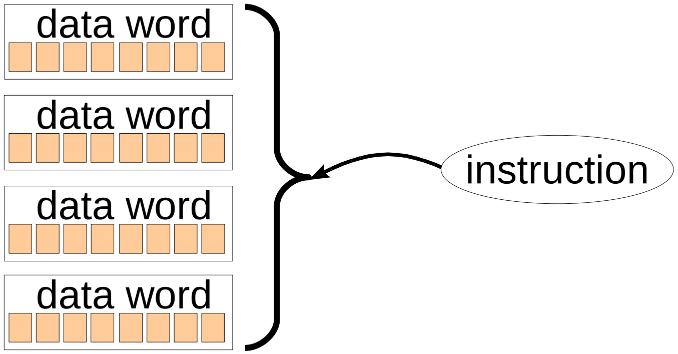
Memory elements and processing units are connected by buses.
All of these are controlled in some way by a sequence of instructions – a program.



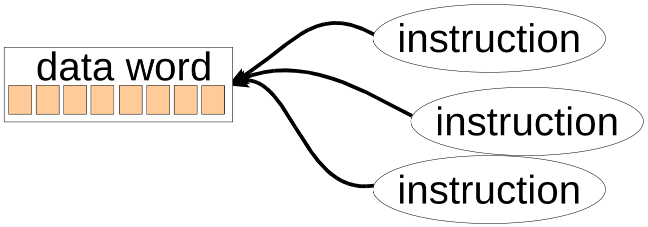
Flynn's Taxonomy



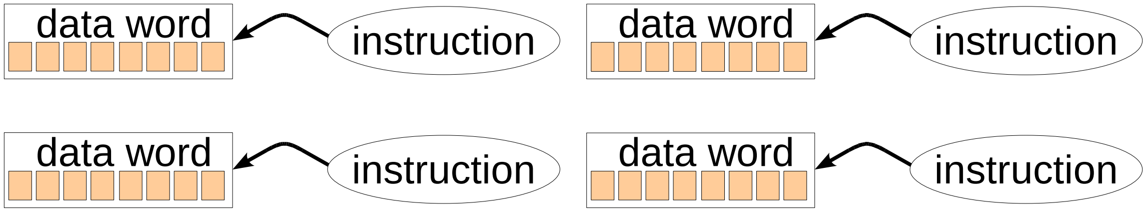
Single Instruction stream, Single Data (SISD)



Single Instruction stream, Multiple Data (SIMD)

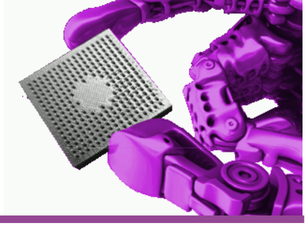


Multiple Instruction stream, Single Data (MISD)



Multiple Instruction stream, Multiple Data (MIMD)

Types of architecture



Von Neumann architecture

The same memory holds both data and instructions, both are transferred to the CPU through the same buses.

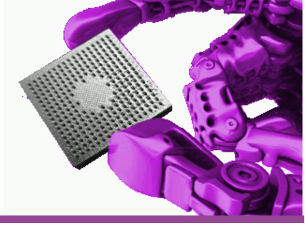
Harvard architecture

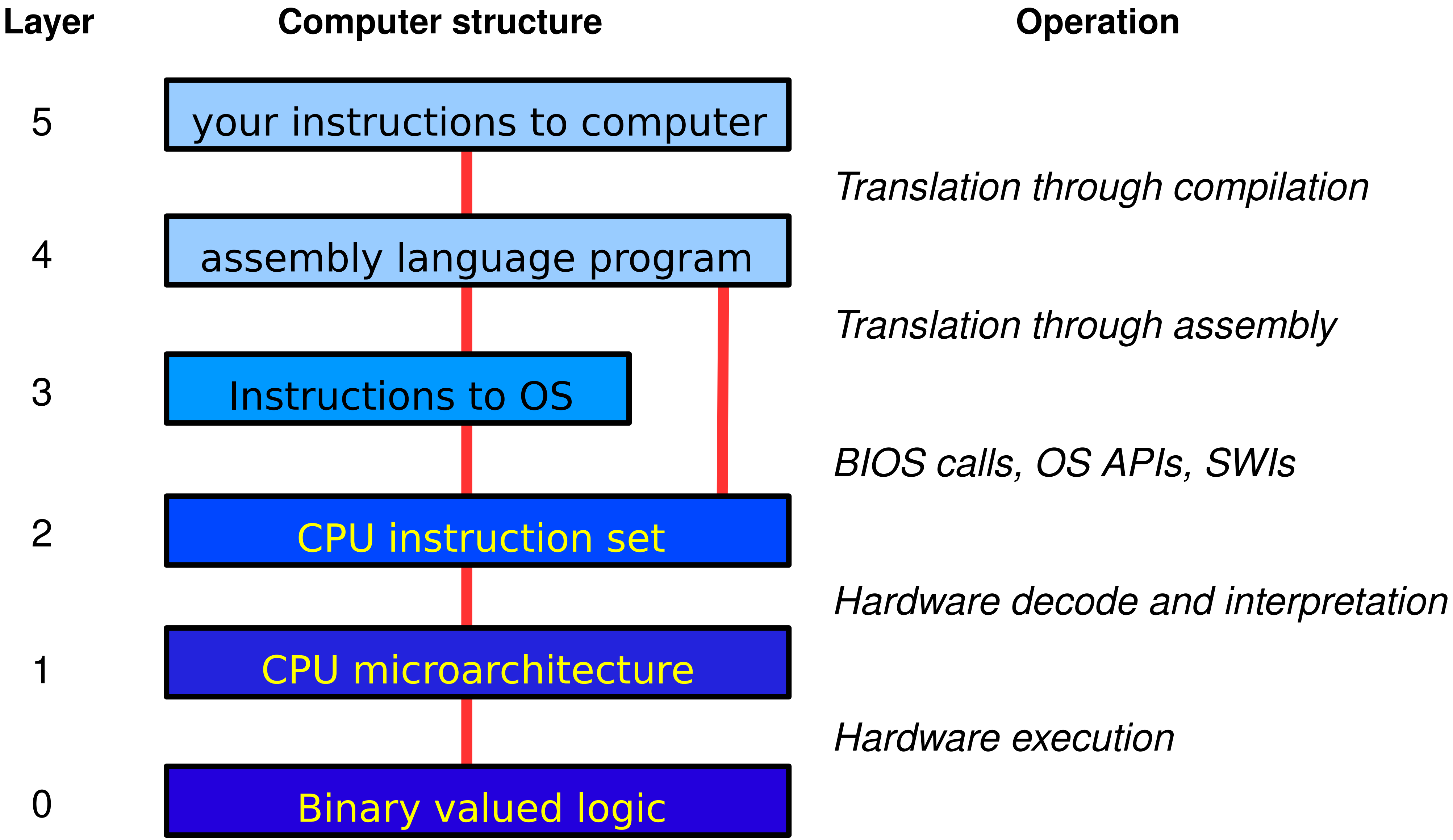
One memory holds data and another one holds instructions, each are transferred to the CPU through separate buses.

Other architecture

Many alternatives. For example, several dedicated buses (ADSP2181), shared address and separate data buses, or separate address buses and shared data bus.
Cache memory may provide the CPU with an internal Harvard architecture, but external von Neumann buses (or vice versa)!

Layers of computation





Layer

Computer structure

Operation

5 your instructions to computer

Translation through compilation

4 assembly language program

Translation through assembly

3 Instructions to OS

BIOS calls, OS APIs, SWIs

2 CPU instruction set

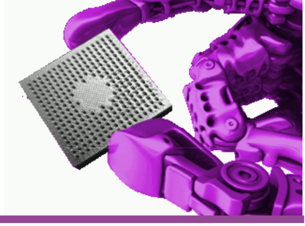
Hardware decode and interpretation

1 CPU microarchitecture

Hardware execution

0 Binary valued logic

A few definitions



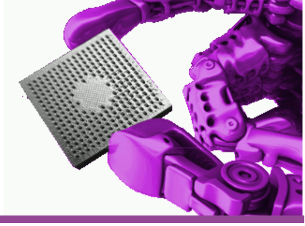
Q. What is an ALU?

Arithmetic Logic Unit - can perform simple arithmetic and logic operations such as add, subtract, NAND, OR, etc...
Input and output are usually from/to registers, connected through a fast bus. The ALU deals with fixed point numbers only, and usually operates within a single instruction cycle.

Q. What is an FPU?

Floating Point Unit - either on-chip, or an external co-processor, it performs complex arithmetic on floating-point numbers (usually in IEEE 754 format).
FPUs are usually very slow (may take hundreds of instruction cycles to calculate), and takes its I/O direct from special floating point registers that the CPU can write to.

A few definitions



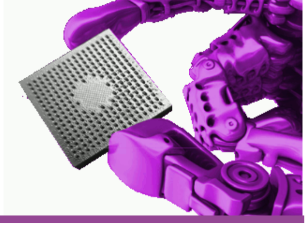
Q. What is an MMU?

Memory Management Unit - if you need to use virtual memory then this unit translates an address that the processor needs to access into a real address in memory. The processor just sees a large continuous address space of memory, while the MMU hides the real memory organisation.

Q. What is MMX/SSE?

Multimedia Extensions/Streaming SIMD Extension - an on-chip co-processor from Intel (also AMD and others) designed to speed up some multimedia-based operations by allowing simple calculations to be done on different items of data (usually integers) in parallel.

A few definitions



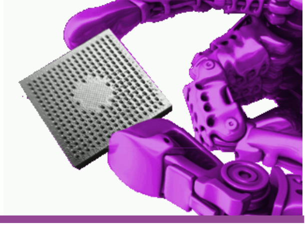
Q. What is a register?

Data/address/control register - on-chip memory locations that are directly wired to internal CPU buses to allow extremely fast access (within one instruction cycle). The distinction blurs with on-chip memory for some CPUs, and with the stack in processors like the picoJavaII.

Q. What is a clock/instruction cycle?

Instruction cycle - the time taken to fetch an instruction, decode it, process it and return the result. This may be one or more periods of the main clock cycle (derived from an external oscillator).
For RISC processors, instructions typically execute in a fixed number of clock cycles (often in one cycle).
For CISC processors, some instructions can take a lot longer to process.

A few definitions



Q. What is a RISC CPU?

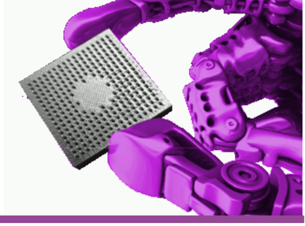
Reduced Instruction Set Computer - any CPU is limited by its slowest component and its size. Based on the premise that *80% of instructions use only 20% execution time, and the remaining 20% use up 80% of the chip area*, CPUs were reduced to contain the 80% most useful instructions. Sometimes RISC means <100 instructions.

Q. What is a CISC CPU?

Complex Instruction Set Computer - think of any useful operation and directly insert this into the CPU hardware. Don't worry how big, power hungry, or slow this will make the CPU. Early VAXs had instructions that reportedly took more than 2000 clock cycles to execute!

Some modern processors emulate a CISC instruction set with a RISC core.

A few definitions



Q. What is big endian?

Big endian - most significant byte first, as used by 68000, SPARC etc..

Q. What is little endian?

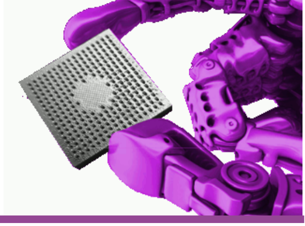
Little endian - least significant byte first, as used by Intel 80x86 family.

Some processors (such as the ARM) allow for switchable *endiness*.

To tell the difference, consider the way a multi-byte word (such as a 32-bit integer) is represented. Little endian would have it start with the least-significant byte, then the next, then the next and finally the most significant byte.

Any other ordering apart from this is big endian.

Binary Formats & Arithmetic



Binary number formats

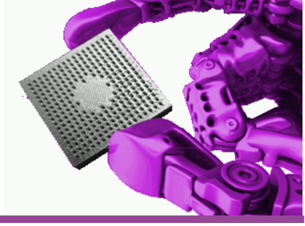
Everything stored in a computer is represented as strings of 1's and 0's – binary numbers. However the way we interpret those bits, and the number of bits used to store a particular value will vary depending upon *what* we are trying to store.

A single letter of the English alphabet can be stored as a character: 8-bits, as an unsigned 2's complement byte.

However the precise distance from the Earth to the Sun in millimetres is so big it would need at least 64-bits of storage space. If it is not a precise integer, then we would need to use a fractional or floating point format to store the number.

The way numbers are stored is important; it determines how much memory you need, how difficult it is to write your programs, how fast the computer operates, and how much power it consumes...

Fixed point numbers



Unsigned binary numbers

Bit weightings reading from the right are 1, 2, 4, 8, 16...

e.g. 00101001 = 32 + 8 + 1 = 41d

(we use a 'b' and a 'd' to represent binary and decimal, i.e. 41d)

† *A great format for calculating machines (so get used to it!)*

† *Doesn't represent any negative value.*

Sign-magnitude

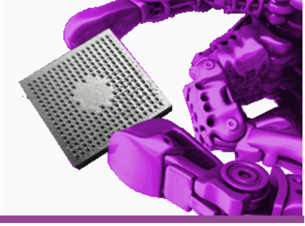
MSB is the sign bit (-ve if MSB=1), other bits as for unsigned binary

e.g. 00101010b = 42d   and 10101010b =-42d

† *Relatively easy for humans to read*

† *Complicated for machine to do arithmetic*

Fixed point numbers



Excess-n

Any number n is stored as $n+2^{(m-1)}$
For example, excess-127 can be used for 8-bit numbers, where a number is stored as itself plus 127.
So numbers from -127 to +128 can be stored.
e.g. 10101110b = 47d and 01010010b = -45d

> We will use this later in IEEE 754 floating point representation

> A little confusing for humans (what you see is *not* what you get!)

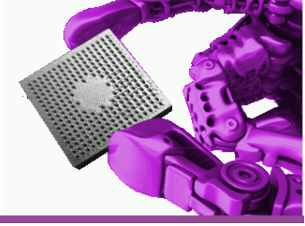
Binary coded decimal

Used in early computers, each digit (0-9) is coded in binary as a separate nibble.
e.g. 0011 0101b = 35d and 0100 0001b = 81d

> Very easy to read by humans, easy arithmetic

> Wastes space (4 bits can hold 0-16, but each digit is 0-9)

Fixed point numbers



One's complement (now very rarely used)

MSB is sign bit (-ve if MSB=1)

Other bits hold magnitude (in unsigned format) BUT

If number is -ve, the binary digits are all swapped.

e.g. 00101100b = 44d and 11010011b = -44d

Two's complement

MSB is sign bit (-ve if MSB=1)

For -ve numbers, just take ones compliment and add 1

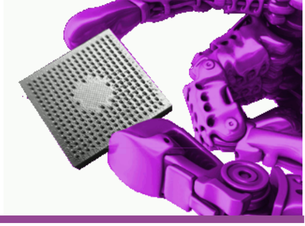
e.g. 00101101b = 45d and 11010011b = -45d

For addition, use standard unsigned binary arithmetic.

> Very easy to perform arithmetic

> Slightly harder for a human to read

Q-format fractional numbers

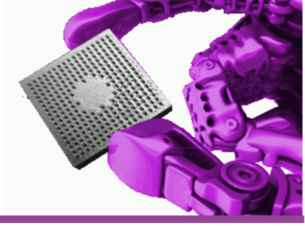


Q*y*-format fractional numbers are often called “(x.y) format” (where x+y= total no.of bits in the word)

To represent a fraction in fixed point binary arithmetic, just move the logical position of the radix point. For example, here are some of the possible 16-bit Q formant numbers:

Format	binary sequence	decimal range
Q0 (16.0)	0000000000000000	-32768 to 32767
Q1 (15.1)	0000000000000000	-16384 to 16383.5
Q2 (14.2)	0000000000000000	-8192 to 8191.75
Q7 (9.7)	0000000000000000	-256 to 255.9921875
Q15(1.15)	0000000000000000	-1 to 0.9999694824..

Q-format fractional numbers



- Q-format allows fractions to be represented in fixed point, and standard arithmetical operations to be carried out.
- The numbers have a fixed error (quantization) size which can be very big as a percentage error when the number value becomes small.

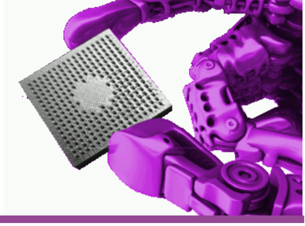
Q9 (7.9) 0001 1001 0100 0000

64 32 16 8 4 2 1 0.5 0.25 0.125

value = 8 + 4 + 0.5 + 0.125 = 12.625

Q-format is useful in Digital Signal Processing.

But for now, we will concentrate on standard integer two's complement format (which is the same as Q0 or 16.0) .



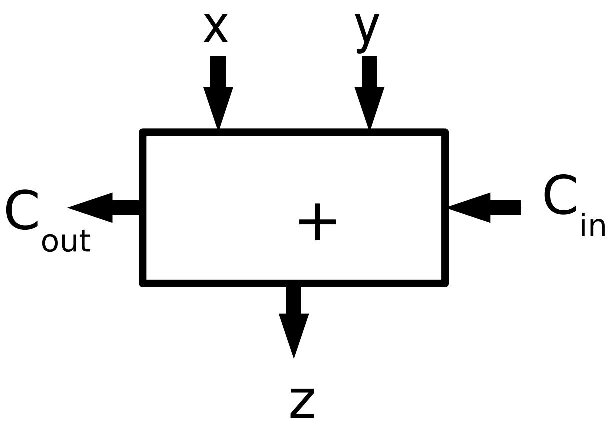
Addition and subtraction

Assuming two's complement numbers

For each 2 bits to add, generate a sum and a carry
Any carry generated from the most significant bits is discarded

Half Adder - add 2 single bits. Gives result + carry.

Full Adder - add 2 single bits and a carry. Gives result + carry.



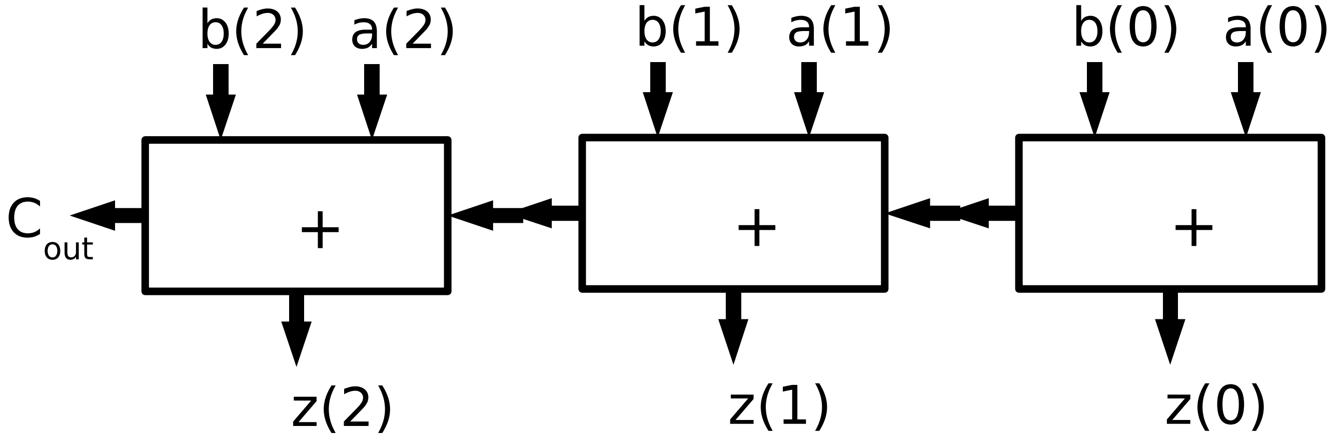
$z=x\oplus y\oplus c_{in}$

$z=xyc_{in} + xyc_{in} + xyc_{in} +xyc_{in}$

$c_{out}=xy + xc_{in} + yc_{in}$

Addition and subtraction

The full adder can be used as a serial adder to add 2 bits every clock cycle (and therefore 32 bits in 32 clock cycles), or as a faster ripple carry (or parallel) adder:



Looks OK? But remember the carry out equation?

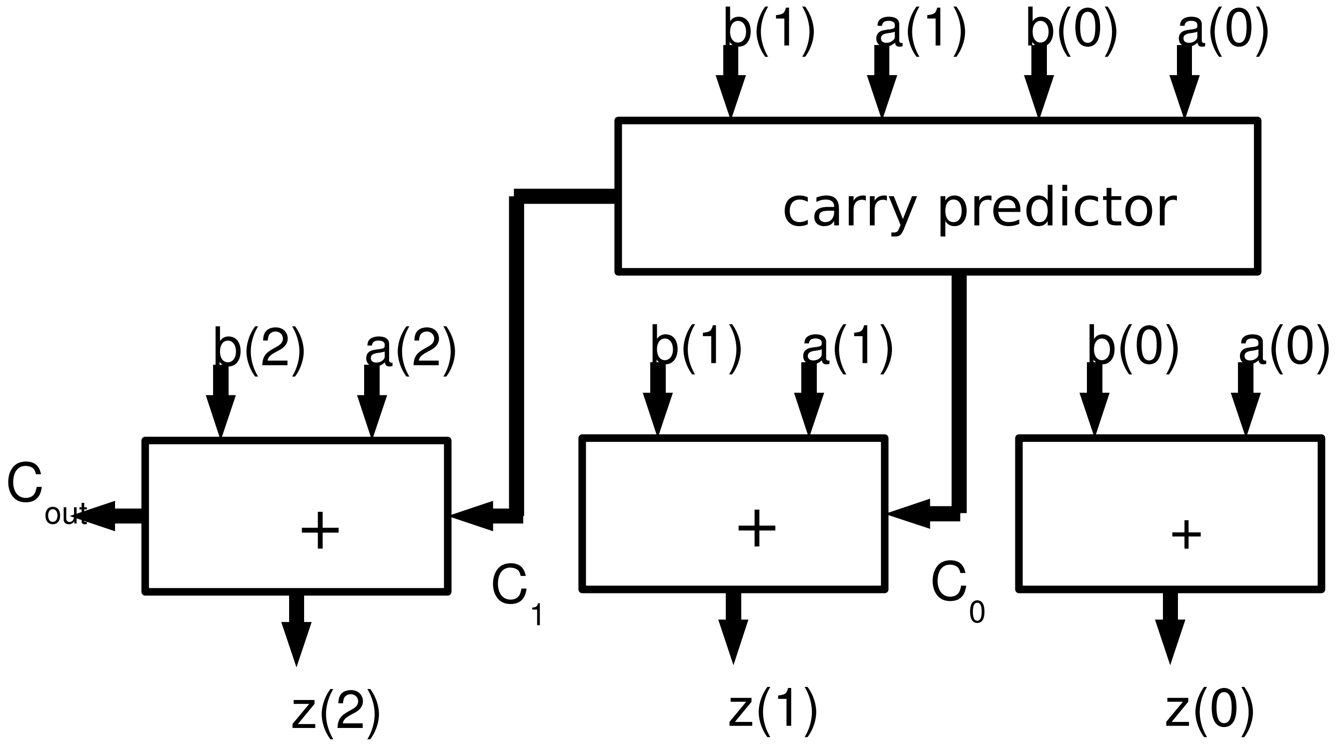
$$c_{out}=xy + xc_{in} + yc_{in}$$

Need to wait once for AND and once for OR to get every C_{out}, and this has to also wait for each C_{in}. So the n bit adder needs to wait a total of *2n* propagation delays until output z(n) appears.

Addition and subtraction

The solution is to use **carry look-ahead**

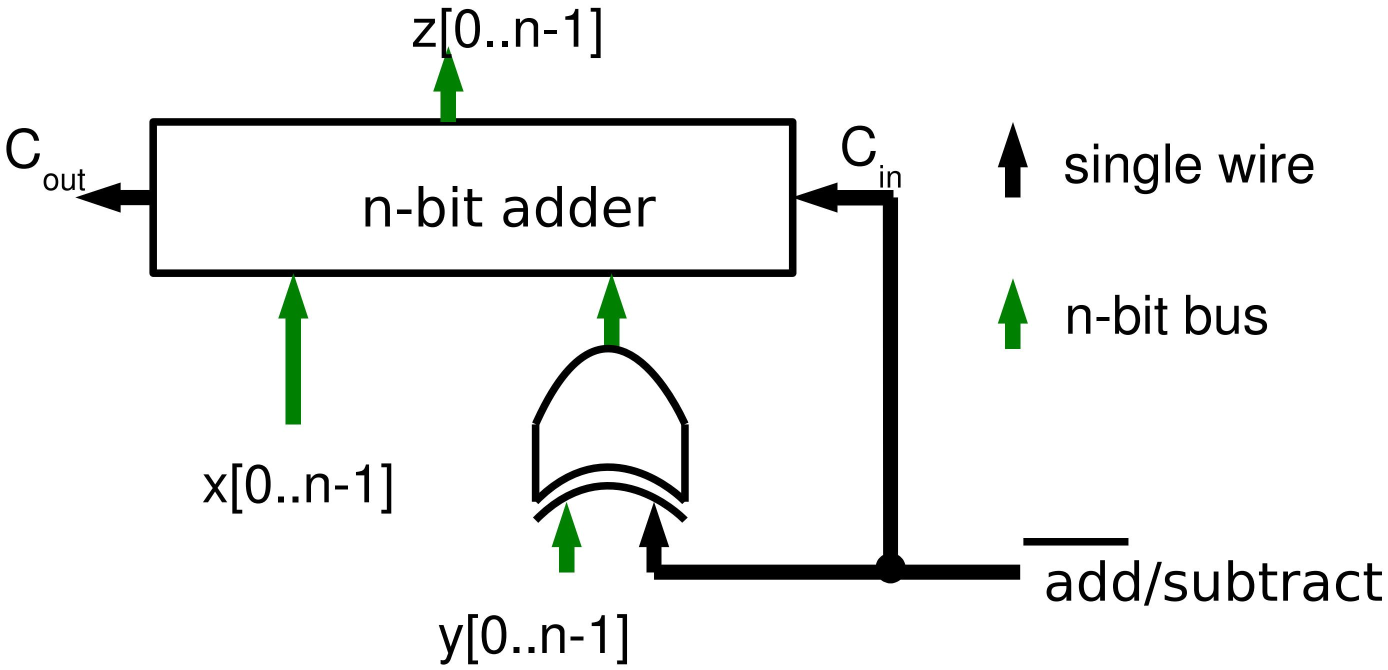
This adder is larger and more complex, but is much quicker.



The carry bits can be generated with only *2* propagation delays (not *2n*)

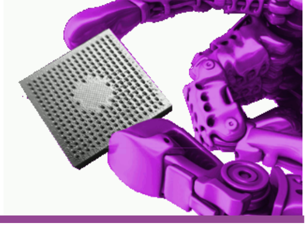
Addition and subtraction

To subtract with two's complement, just make one of the inputs negative (*to do this, you need to flip the bits & add 1 to it*), and use a normal adder.



For this adder, can we use C_{out} to set an overflow flag?

Addition and subtraction



More about overflow....

consider some two's complement arithmetic examples:

```
0010 + 1110 = ?
2   +   (-2)  = ?

0010 + 1110 = 0000 + carry
```

in two's complement
in decimal

answer is 0 but why a carry?

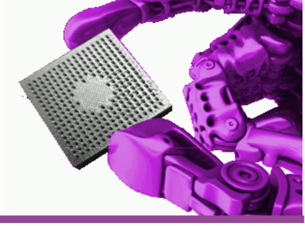
```
0111 + 0110 = ?
7   +   6 = ?


0111 + 0110 = 1101
```

but 1101=-3 !! Should be 13
Answer is wrong but there's no carry!

Solution is to compare the sign bit of the operands before the addition. If the sign bits are opposite, then no overflow can occur. If they are the same, then the result should have same sign (and if not, this means an overflow has occurred).

Multiplication



Many methods to do *m*×*n* including:

Add *m* together *n* times [use an adder and a counter -

- Sum of (shifted) partial products [not quick, but is simple]

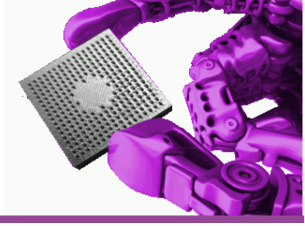
- Split into adds and left shifts [fast, but different calc. times]

- Booth multiplier, and Robertson's method [fast but complex]

- Convert to logs and add [needs big lookup table]

- Use an alternate number format [an active research area]

Multiplication



Partial products

Worked example: 9 × 11 (unsigned)

```
A        1001     multiplicand (9)
B        1011     multiplier (11)
     ________
         1001
        1001      partial products
       0000
      1001
     ________
C     1100011     result (99)
```

Recipe:

```
C=0
loop i=0..3
    if A[i]=1
        C=C+B<<i
```

Ingredients:

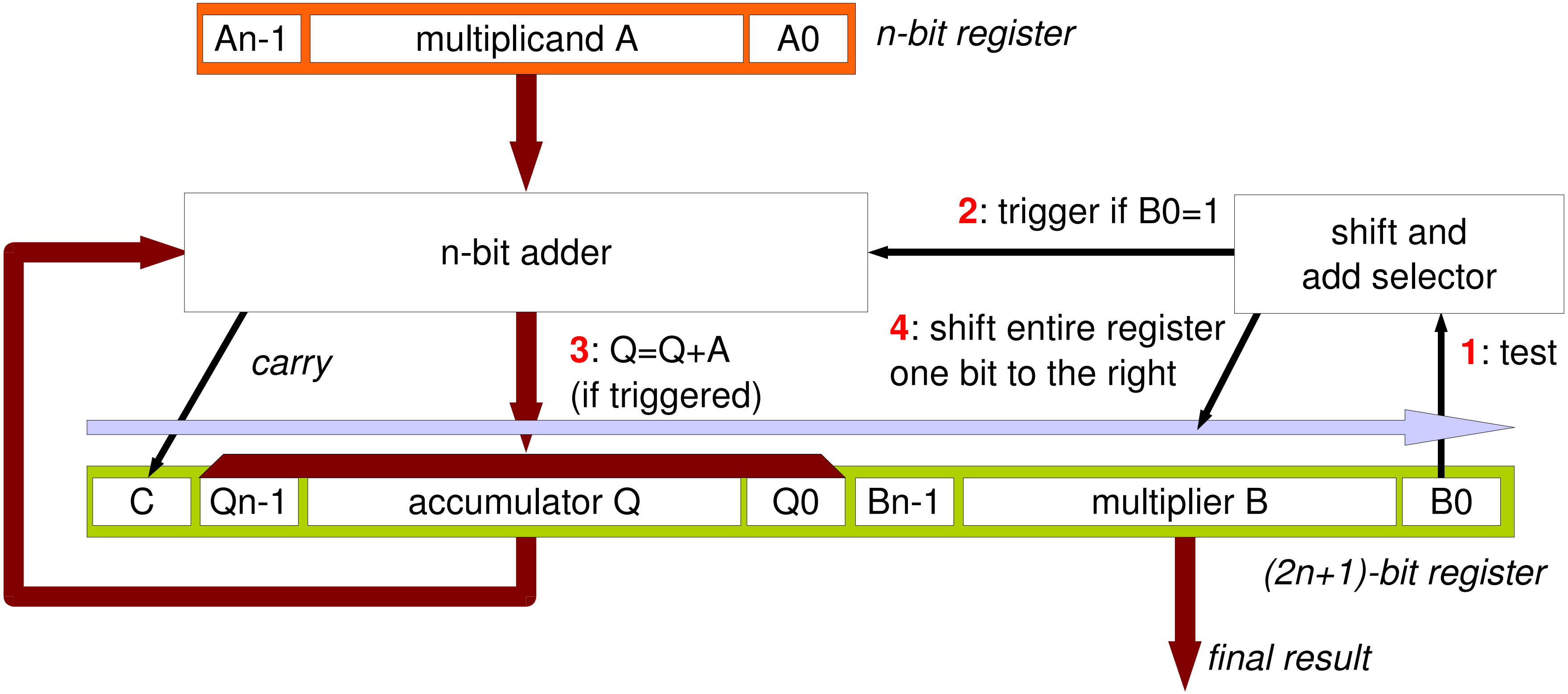
1 n-bit input register (A)
1 2n-bit input register (B)
1 shifter
1 2n-bit adder
1 2n-bit output register

Duration:

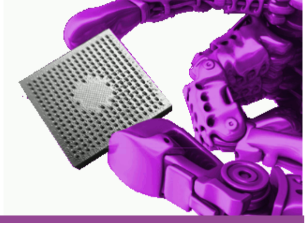
n loops of (compare|add)

Multiplication

Block diagram of partial product method:



Multiplication



Booths algorithm

Look at every bit in the multiplier and compare to its neighbour (in turn).

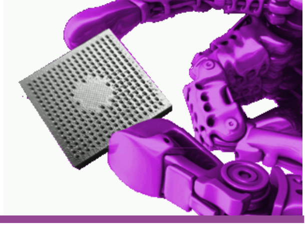
For each pair, either add, subtract or ignore the partial product in this shift position according to the rule:

$X_i X_{i-1}$	Rule
0 1	Add multiplicand (shifted by i)
1 0	Subtract multiplicand (shifted by i)
0 0	Do nothing
1 1	Do nothing

Worked example: (unsigned)

```
A                0000 0000 0001     multiplicand
B                0111 1011 1100     multiplier
             ------------------
              00 0000 0000 01     -   B2B1=10 so -A<<2
         00 0000 0000 01          +   B6B5=01 so +A <<6
        000 0000 0000 1           -   B7B6=10 so  -A <<7
   000 0000 0000 1                +   B11B10=01 so +A <<11
  -----------------------------
C  000 0000 0000 0111 1011 1100     result
```

Multiplication



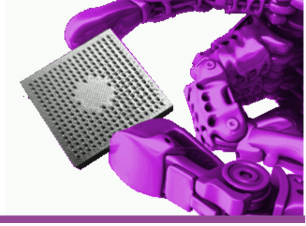
Instead of two bits at a time, we can lump into 4 or 8 adjacent bits (Booth & Robertson's).

These methods can be very fast, but are complex to design. In fact a 32-bit Booths multiplier would be larger than the silicon of the entire basic ARM CPU! That is why the original ARM has no Booths multiplier, and its standard multiply instruction takes many cycles to complete – calculating a set of partial products.

Many microcontrollers have no multiply instruction, however all DSP processors do. E.g. the ADSP2191has a 40-bit dedicated MAC (multiply-accumulate) unit that operates in a single cycle.

In fact the MAC unit in the ADSP2191 also includes a second add/ subtract function (there is another one in the separate ALU, and even a third one in the address register unit!).

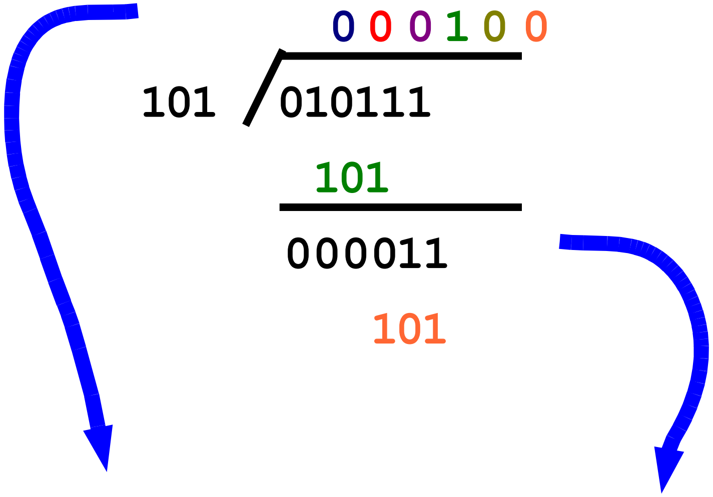
Division



Many CPUs do not have division instructions... modern x86 CPUs do but the ARM and PIC don't. The ADSP2181 has a divide instruction that must be executed 16 times in order to do 16-bit division... in fact this is how we do n-bit division: as n compare/subtracts:

Worked example: (unsigned)

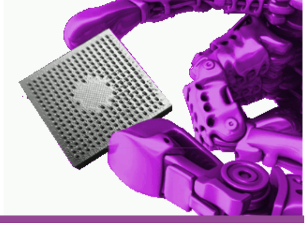
010111 ÷ 101



```
         0 0 0 1 0 0
101 / 010111
       101
      ------
      000011
         101
```

```
Answer is 000100 remainder 11
         23÷5 = 4 remainder 3 ✔
```

Division



So to do the n-bit division of $D \div V$ the algorithm is;

1. set $i=n$, set $R=D$
2. compare $V<<i$ with R, decrementing i until $(V<<i) \geq R$
3. then set $R=R-(V<<i)$ and put a 1 in the quotient (answer word in bit position i) and repeat until $i=0$.

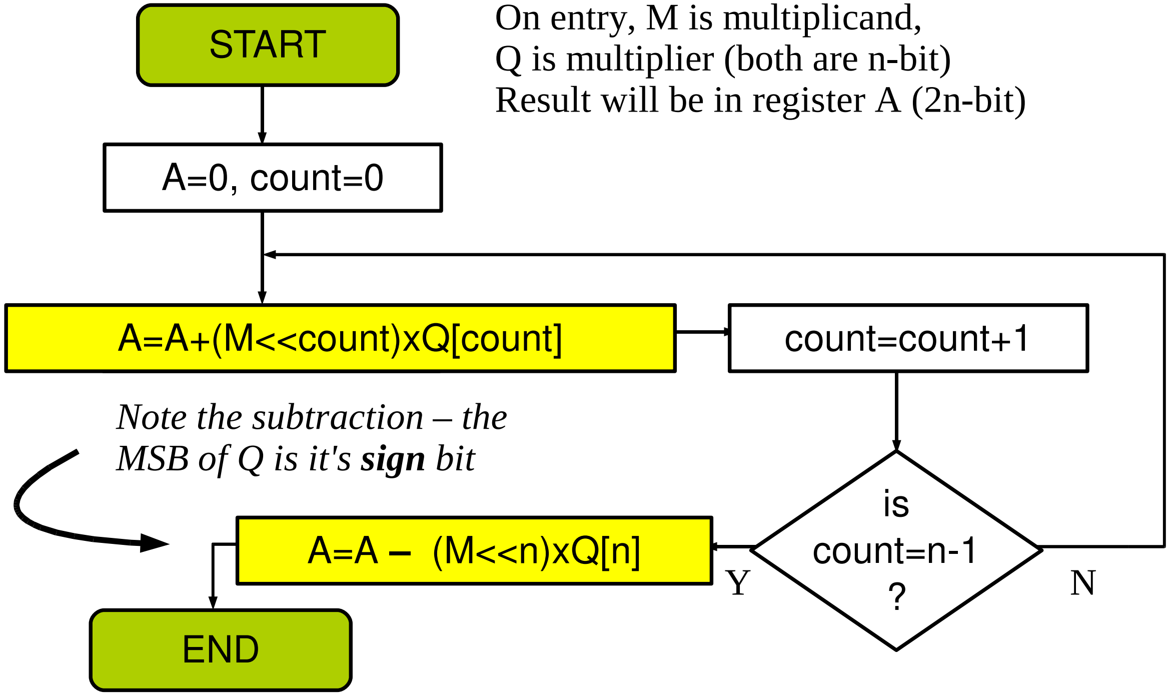
4. At the end, the quotient holds the answer, with the remainder in R

For n-bit signed integers:

1. first take two's complement of any negative numbers to make them positive
2. do an $n-1$ bit division (i.e. ignore the sign bits)
3. if the input signs were different, the answer sign is negative
4. if the input signs were the same, the answer sign is positive

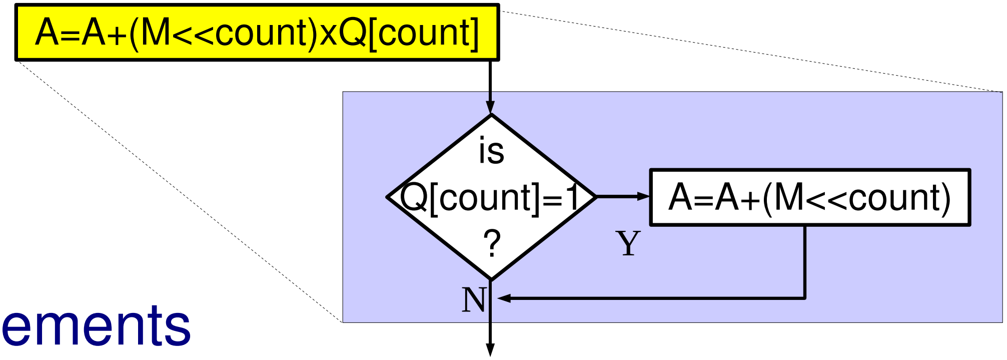
Division

On entry, M is multiplicand,
Q is multiplier (both are n-bit)
Result will be in register A (2n-bit)



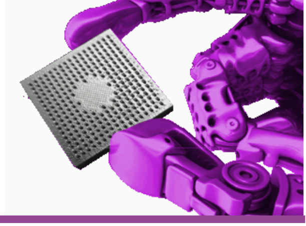
Note the subtraction – the MSB of Q is it's ***sign*** *bit*

There is no actual 1-bit multiplication in the yellow boxes, only a switch decides whether to accumulate or not:



How many adds, subtracts, dec/increments for 16 bit addition? Are these constant?

Q-format arithmetic



Remember the Q-format? The logical position of the radix point must be taken into account by the programmer (*this is not a hardware issue!!!*) for arithmetic operations:

Q-format addition/subtraction:

The Q-format of both numbers must be the same for this to work...

`0111 + 0101 = 1100`

Q2 (2.2) + Q2 (2.2) = Q2 (2.2)

`1.75 + 1.25 = 3.00`

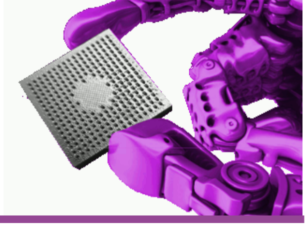
`0111 + 0101 = 1100`

Q2 (2.2) + Q3 (1.3) = Q????

`1.75 + 0.625 = ?.??`

What Q-format is the result? We know the answer is 2.375, but how can binary number 1100 equal this?

Q-format arithmetic



Q-format multiplication:

The Q-format of both numbers can differ, but must be known:

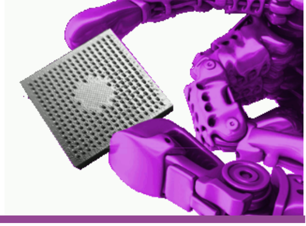
`0111 × 0101 = 0010 0011`

Q2 (2.2) × Q3 (1.3) = Q5 (3.5)

`1.75 × 0.625 = 1.09375`

What Q-format is the result? If we multiply numbers of format (n.m) and (p.q) then the resultant format is (n+p.m+q), of course the multiply result is always twice as long as the operand length. This is normal for every binary multiplier. Q-format arithmetic uses exactly the same hardware as normal arithmetic.

Q-format arithmetic



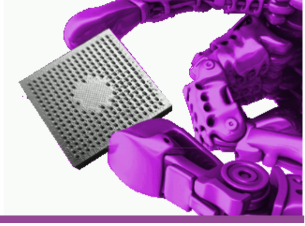
Q-format multiplication (continued)**:**

One standard multiply is Q15 × Q15 or (1.15) × (1.15). Of course the result is a (2.30) number. If we shift this <u>left</u> by one bit we get a (1.31) result. Then the top word is (1.15), the same format as the operands.

The Q15 format has a maximum value of 1.0, and so the multiply result can never be greater than 1.0 (1 × 1). Using this format can guarantee no overflow!!

Many CPU instruction sets (such as ADSP2181) have a multiply & left shift instruction that is designed to make Q15 multiplication easy. Just take the top word of the multiply result...

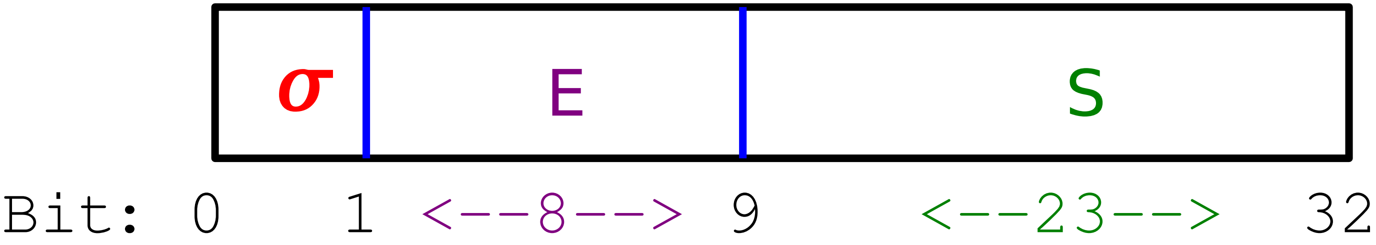
Floating point



Floating point extends the range of numbers that can be stored by a computer, and the accuracy is independent of the number magnitude.

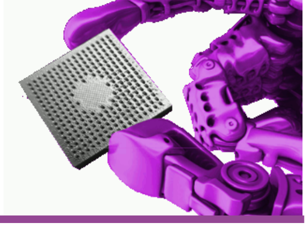
Floating point numbers store a sign (σ), exponent (*E*) and mantissa (*S*) and are of predetermined number base (B). B is usually set to base-2.

$$\text{floating point value} = (-1)^{\sigma} \times S \times B^{E}$$



E and S are both fixed point 2's complement unsigned numbers

Floating point



We will now consider IEEE standard 754 floating point representation. This is used in almost all modern FPUs.

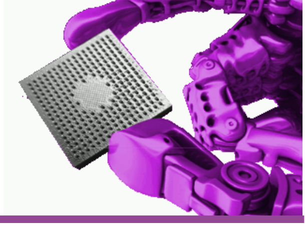
IEEE754 has 1 sign (σ), 8 exponent (E) and 23 mantissa bits (S) for single-precision numbers. Double precision is 1, 11 and 52 bits.

The exponent is stored in excess 127 for single precision (and excess 1023 for double precision).

The mantissa is slightly unusual; the 23 (52) bits are in Q23 (Q52) format and it is assumed that 1 must be added to the mantissa, but the 1 is not stored. In other words, the mantissa is a fractional value ranging from 1 (mantissa bits 0) to slightly below 2 (all mantissa bits 1).

As the exponent is a power of 2, the mantissa never needs to be greater than 2 (you would just add 1 to the exponent instead).

Floating point

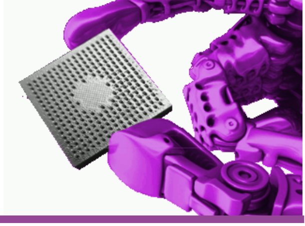


IEEE standard 754 floating point

There are 5 different number types:

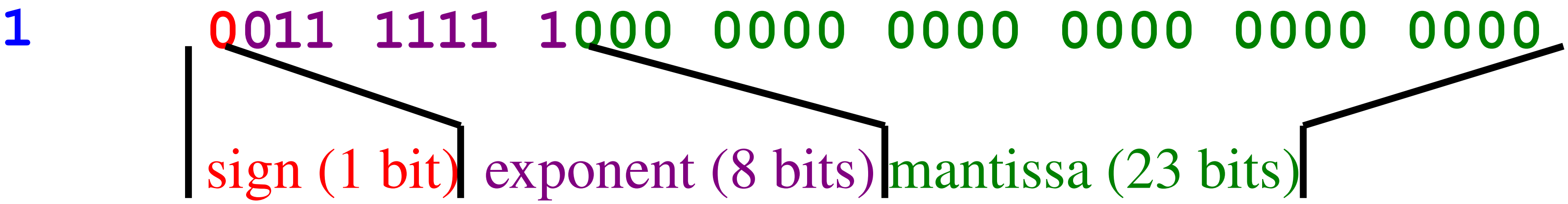
Zero	±	0	0
Infinity	±	111...111	0
NaN	±	111...111	non 0
Denormalised	±	0	any non-zero sequence
Normalised	±	E != 0	anything

Floating point



IEEE standard 754 floating point

Most numbers fall in the normalised range. Here are some examples:



```
1       0011 1111 1000 0000 0000 0000 0000 0000
```

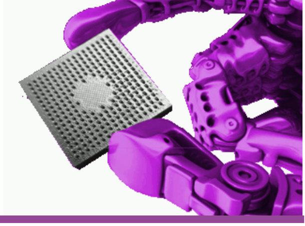
sign (1 bit) exponent (8 bits) mantissa (23 bits)

This is positive, has exponent of 127-127=0 (remember it's in excess-127 format), and mantissa of 1 so 1×2^{0} $(S\times2^{E}) = 1.0$

```
-40     1100 0010 0010 0000 0000 0000 0000 0000
```

This is negative, has exponent of 132-127=5, and mantissa of 1.25 so $-1\times1.25\times2^{5}$ $(\sigma\times S\times2^{E}) = -40.0$

Floating point

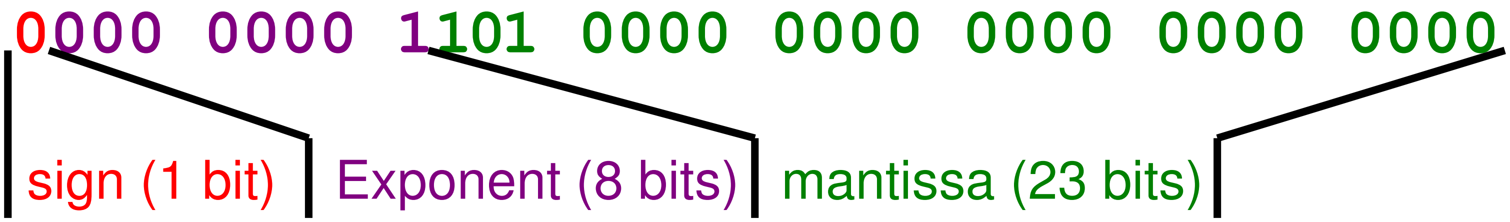


IEEE standard 754 floating point

Denormalised numbers are smaller than normalised numbers. They do not have the same implicit "1." in the mantissa - it is instead just a straightforward binary fraction with maximum size just below 1. This fraction must be assumed to be multiplied by the smallest normalized number:

(which is exponent 1, mantissa 1 = 1.0×2^{-126}).

Here is an example denormalised number:

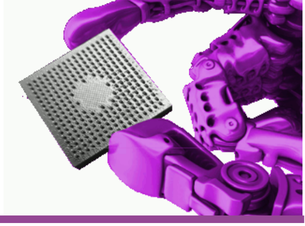


```
0000 0000 1101 0000 0000 0000 0000 0000
```

sign (1 bit) | Exponent (8 bits) | mantissa (23 bits)

This is $+1\times2^{-126}\times(0.5+0.125) = 7\times10^{-39}$

Floating point

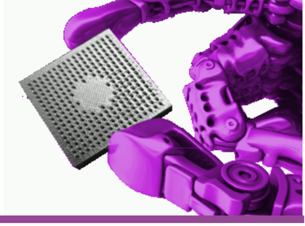


To summarise, IEEE 754 single precision numbers are 32 bits long, double precision are 64 bits long.

Special bit-patterns represent 0, NaN and infinity (and allow for very small numbers to be stored in denormalised mode).

Finally, special extended intermediate formats allow overflow to occur during a calculation, as long as the result does not overflow, this improves the accumulation of round-off errors through a calculation.

Extended single precision is 43 bits for single precision (1+11+31) and 64 bits for extended double precision (1+15+63)



Floating point arithmetic

Addition and subtraction of general floating point

First shift numbers so their exponents are the same, then add mantissas:

To do this addition:

$$
\begin{array}{rl}
 & 0.824 \times 10^{2} \\
+ & 0.992 \times 10^{4}
\end{array}
$$

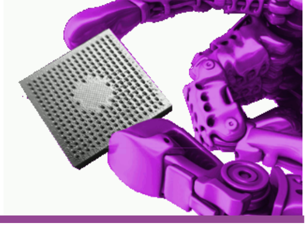
Do we shift up...

$$
\begin{array}{rl}
 & 0.00824 \times 10^{4} \\
+ & 0.992 \quad \times 10^{4}
\end{array}
$$

or down?

$$
\begin{array}{rl}
 & 0.824 \times 10^{2} \\
+ & 99.2 \quad \times 10^{2}
\end{array}
$$

Floating point arithmetic



Addition and subtraction:

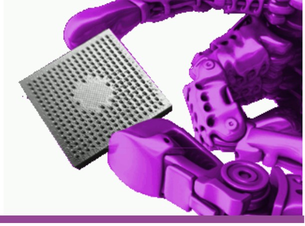
1. Equalise exponents
2. Add/subtract mantissas
3. Normalise the result

Remember that the mantissa has a *hidden bit*

Must check for zeros in input or shifted values, or after adding the mantissas - because zero has a unique IEEE754 bitpattern.

If the added mantissas overflow, then normalise, shift and increment/decrement the exponent (for add/subtract)

Floating point arithmetic



Multiply and divide:

1. Add (for multiply) or subtract (for divide) the exponents
2. Multiply/divide the mantissas
3 Normalise the result

$$(0.824\ \mathrm{x}10^{2})\ \mathrm{x}\ (0.992\ \mathrm{x}10^{4})=?$$

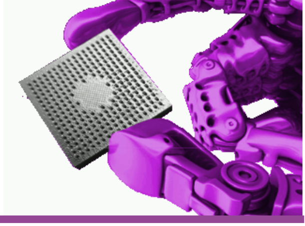
$$(0.824\ \mathrm{x}\ 0.992\ )\ \mathrm{x}10^{(2+4)}=0.817408\ \mathrm{x}10^{6}$$

$$(0.824\ \mathrm{x}10^{2})\ \div\ (0.992\ \mathrm{x}10^{4})=?$$

$$(0.824\ \div\ 0.992\ )\ \mathrm{x}10^{(2-4)}=0.8306..\ \mathrm{x}10^{-2}$$

And remember to check for zeros, consider hidden bits and over/underflow!

Floating point arithmetic



Guard bits:

Some floating point units operating on n-bit values have a width of more then n bits. Guard bits are in the least significant positions to reduce the effects of rounding, and ensure that the result is accurate to n-bits.

Subtraction

$$\begin{array}{rl} & 1.0000\ 0000 \quad \times\ 2^{1} \\ - & 1.1111\ 1111 \quad \times\ 2^{0} \end{array}$$

Equalise exponents

$$\begin{array}{rl} & 1.0000\ 0000 \quad \times\ 2^{1} \\ - & 0.1111\ 1111 \quad \times\ 2^{1} \end{array}$$

Answer

$$\begin{array}{rl} = & 0.0000\ 0001 \quad \times\ 2^{1} \end{array}$$

guard bit

Subtraction

$$\begin{array}{rl} & 1.0000\ 0000\ 0 \quad \times\ 2^{1} \\ - & 1.1111\ 1111\ 0 \quad \times\ 2^{0} \end{array}$$

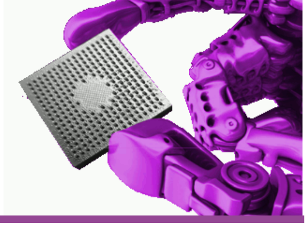
Equalise exponents

$$\begin{array}{rl} & 1.0000\ 0000\ 0 \quad \times\ 2^{1} \\ - & 0.1111\ 1111\ 1 \quad \times\ 2^{1} \end{array}$$

Answer

$$\begin{array}{rl} = & 0.0000\ 0000\ 1 \quad \times\ 2^{1} \end{array}$$

Floating point arithmetic



Rounding:

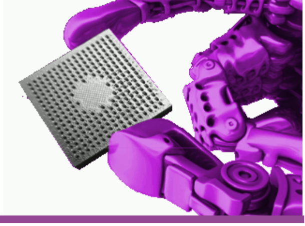
Both floating and fixed-point arithmetic requires rounding. There are four main methods of numeric rounding:

Round to nearest (default) : round to the nearest representable value, and if two values are equally near, default to the one with LSB=0

- Round towards $+\infty$: round towards the most positive number.
- Round towards $-\infty$: round towards the most negative number.

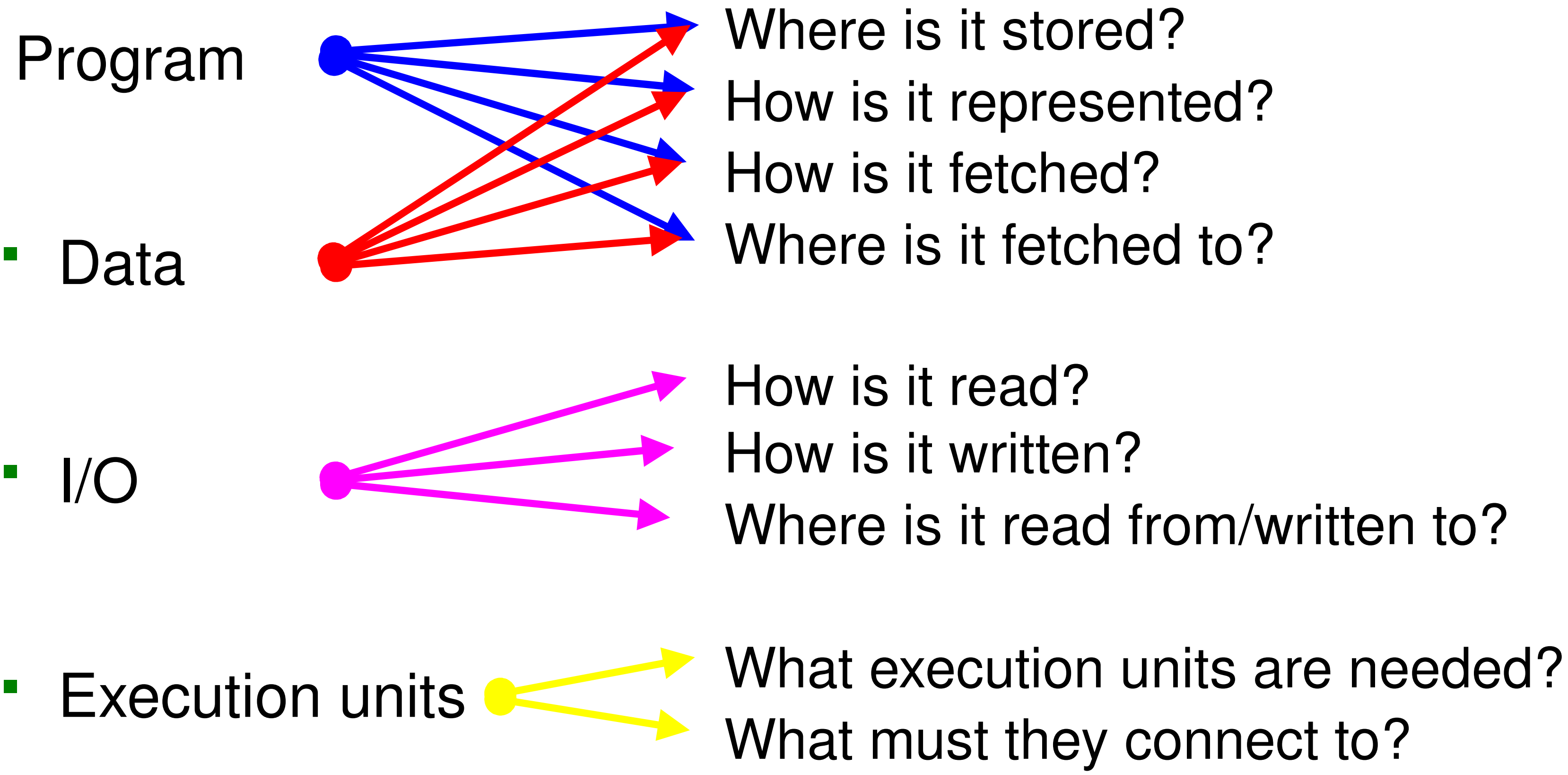
  In practice, you can do each calculation twice, once rounding to $+\infty$ and once to $-\infty$. Then the average of the two is used (and the difference between the two gives the numerical accuracy).

- Round towards 0 : equivalent to always truncating the number.



Computer requirements

Some questions to ask ourselves:



Program

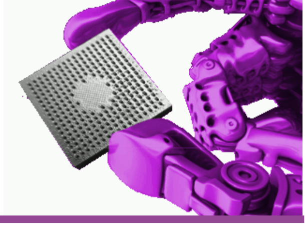
- Data
- I/O
- Execution units

Where is it stored?
How is it represented?
How is it fetched?
Where is it fetched to?

How is it read?
How is it written?
Where is it read from/written to?

What execution units are needed?
What must they connect to?

Computer requirements



We need some method of directing the computer to perform various tasks. This is accomplished by writing a program consisting of sequential instructions (each is selected from an instruction set).

The instructions must be stored. Early computers used punched card, later magnetic tape. Desktop computers started with magnetic disc (floppies and hard disc), and then CD/DVD. Modern computers tend to use flash memory or similar.
Basic ROM is also still used in many systems (it's cheap!)

Programs and data being processed are usually read from their storage devices into RAM before execution.

Buses transport all this data around the computer, on command.

We will meet each of these elements in module 3.