

GPGPU

01

Gábor Valasek (valasek@inf.elte.hu)

References and recommendations

- **Jason Gregory**: Game Engine Architecture (<https://www.gameenginebook.com/>)
 - We follow this for the middle part
- **Jean-Michel Muller**: Handbook of floating-point arithmetic (<https://www.springer.com/gp/book/9780817647056>)
 - For the interested, an excellent reference on floating-point issues
- MIT: Performance Engineering course
 - Both the course recordings and materials are freely available from <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/>
- **Optimization competition (February 2021)**:
 - <https://easyperf.net/blog/2021/02/05/Performance-analysis-and-tuning-contest-4>
 - Hints:
<https://docs.google.com/document/d/1T7OH51gI6me-DU9t-dKU756B3itXtc0GGGihRV-Fab8/edit>

An imaginary computer



Numbers

Unsigned integers

- Most commonly 8, 16, 32, and 64 bits
- Encode the number in binary and truncate

Decimal: 348

Binary: b101011100

Hexadecimal: 0x15C

Unsigned integers

- In N bits the largest representable number is $2^N - 1$, i.e. for
 - 8 bits: $b1111'1111 = 0xFF = 255 = 2^8 - 1 = 256 - 1$
 - 16 bits: $b1111'1111'1111'1111 = 0xFFFF = 65535$
 - 32 bits: $b1111'1111'1111'1111'1111'1111'1111'1111 = 0xFFFFFFFF = 4'294'967'295$
 - 64 bits:
 $b1111'1111'1111'1111'1111'1111'1111'1111'1111'1111'1111'1111'1111'1111'1111'1111'1111 = 0xFFFFFFFFFFFFFFFF = 18'446'744'073'709'551'615$

Unsigned integers

- [illegible]

The following variables are also initialized to the same value:

```
unsigned long long l1 = 18446744073709550592ull; // C++11
unsigned long long l2 = 18'446'744'073'709'550'592llu; // C++14
unsigned long long l3 = 1844'6744'0737'0955'0592uLL; // C++14
unsigned long long l4 = 184467'440737'0'95505'92LLU; // C++14
```


Signed integers

- 1 bit reserved for sign, rest left for the magnitude
- Naive approach: sign bit followed by binary representation of magnitude
 - Symmetric: the smallest and largest numbers in 8 bits are -127 and +127
 - Fly in the ointment! There are two zeros: -0 and +0, e.g. in 8 bits b1000'0000 and b0000'0000
- In practice, mostly use two's complement: (+invert the binary digits and add 1)
 - 8 bit signed integers:
 - 0 to 127 = 0x00 to 0x7F
 - -128 to -1 = 0x80 to 0xFF
 - 32 bit signed integers:
 - 0 to 2'147'483'647 = 0x00000000 to 0x7FFFFFFF
 - -2'147'483'648 to -1 = 0x80000000 to 0xFFFFFFFF

a_{N-1}	a_{N-2}	...	a_1	a_0
-----------	-----------	-----	-------	-------

$$2^{N-1}$$

$$2^{N-2}$$

$$2^1$$

$$2^0$$

a_{N-1}	a_{N-2}	...	a_1	a_0
-----------	-----------	-----	-------	-------

$$2^{N-1}$$

$$2^{N-2}$$

$$2^1$$

$$2^0$$

a_{N-1}	a_{N-2}	...	a_1	a_0
-----------	-----------	-----	-------	-------

$$\leq 2^N - 1$$

Two's complement

a_{N-1}	a_{N-2}	...	a_1	a_0
-----------	-----------	-----	-------	-------

$$\Rightarrow \sum_{i=0}^{N-2} a_i \cdot 2^i - a_{N-1} \cdot 2^{N-1}$$

Two's complement

a_{N-1}	a_{N-2}	...	a_1	a_0
-----------	-----------	-----	-------	-------

 $\Rightarrow \sum_{i=0}^{N-2} a_i \cdot 2^i - a_{N-1} \cdot 2^{N-1}$

- $000\dots 0 = 0$
- $011\dots 1 = 2^{\{N-1\}} - 1$
- $111\dots 1 = -1$

Two's complement

Decimal value	Binary (two's-complement representation)	$(2^8 - n)_2$
0	0000 0000	0000 0000
1	0000 0001	1111 1111
2	0000 0010	1111 1110
126	0111 1110	1000 0010
127	0111 1111	1000 0001
-128	1000 0000	1000 0000
-127	1000 0001	0111 1111
-126	1000 0010	0111 1110
-2	1111 1110	0000 0010
-1	1111 1111	0000 0001

Two's complement

Highest bit is sign: 1 \Leftrightarrow
number is negative

Decimal value	Binary (two's-complement representation)	$(2^8 - n)_2$
0	0000 0000	0000 0000
1	0000 0001	1111 1111
2	0000 0010	1111 1110
126	0111 1110	1000 0010
127	0111 1111	1000 0001
-128	1000 0000	1000 0000
-127	1000 0001	0111 1111
-126	1000 0010	0111 1110
-2	1111 1110	0000 0010
-1	1111 1111	0000 0001

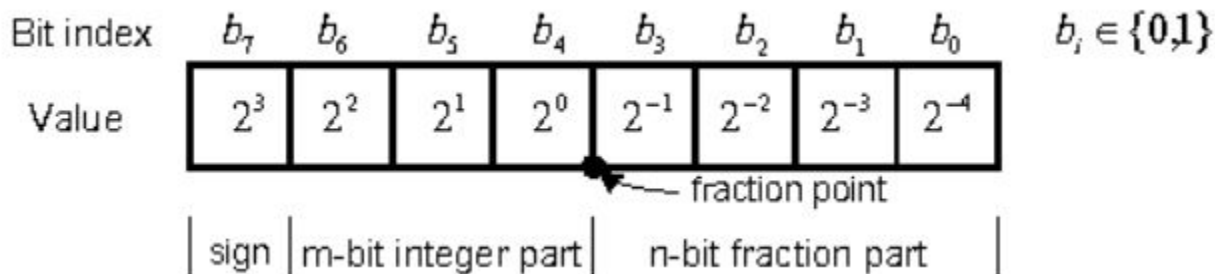
Practically:
invert digits
and add 1

Two's complement

- Many really useful hacks that take advantage of these bit patterns and used in throughout the industry (e.g. $x + \sim x = -1$)
- Even though the C and C++ standards do not specify that your signed integer representation should be two's complement
- On the contrary: they even explicitly list one's and signed magnitude as potential representations (see more in e.g. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0907r1.html>)
- This was a mess! Finally, C++20 ratified that signed integers should be two's complement

Fixed point formats

- Split the bits: n-bit integer and m-bit fractional parts



- It can be extremely useful - but most likely you are going to code some of it
- It has a very limited range; e.g. if you split
32 bits into 1 sign + 16 magnitude + 15 fraction bits,
the largest magnitude you can represent is 65535

Representing real numbers

315.4

Representing real numbers

$$3.154 \cdot 10^2$$

Representing real numbers

$$\textit{integer}.\textit{fraction} \cdot 10^{\textit{exponent}}$$

Representing real numbers

$$\underbrace{integer}_{sign \cdot magnitude} . fraction \cdot 10^{exponent}$$

Representing real numbers

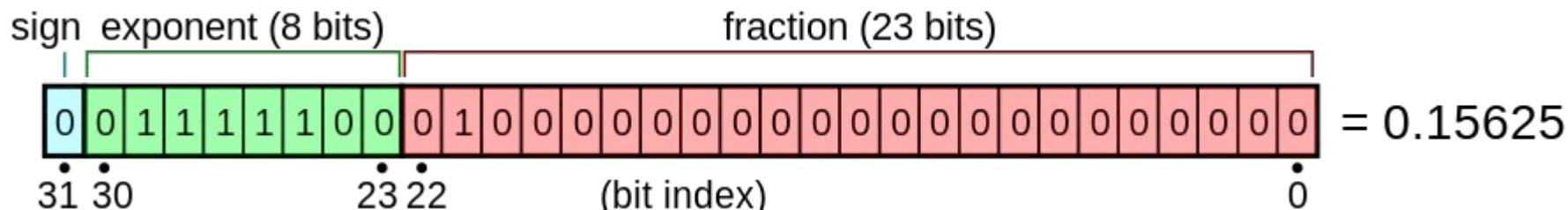
$$\underbrace{integer}_{sign \cdot magnitude} . fraction \cdot 10^{\overbrace{exponent}^{value - bias}}$$

Floating point formats

$$s \cdot 2^{r(e)-bias} \cdot (1 + r(m))$$

Floating point formats

- Let us use a single sign bit s , e exponent, and m mantissa bits
- The value represented by such is $s \cdot 2^{r(e)-bias} \cdot (1 + r(m))$, where $r(e)$ and $r(m)$ is the particular value the current exponent and mantissa bits represent
- As such, we always have an implicit 1 term - the mantissa bits describe the fractional part (if the biased exponent, i.e. $e - bias = 0$), 'always' prepped by 1
- How to split 32/64/whatever bits among the exponent and mantissa? => use a standard, like [IEEE 754-2008](#), e.g. for 32 bits they specify bias 127 and



Floating point formats: [IEEE 754](#)

Name	Common name	Base	Significand bits ^[b] or digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias ^[11]	E min	E max	Notes
binary16	Half precision	2	11	3.31	5	4.51	$2^4 - 1 = 15$	-14	+15	not basic
binary32	Single precision	2	24	7.22	8	38.23	$2^7 - 1 = 127$	-126	+127	
binary64	Double precision	2	53	15.95	11	307.95	$2^{10} - 1 = 1023$	-1022	+1023	
binary128	Quadruple precision	2	113	34.02	15	4931.77	$2^{14} - 1 = 16383$	-16382	+16383	
binary256	Octuple precision	2	237	71.34	19	78913.2	$2^{18} - 1 = 262143$	-262142	+262143	not basic

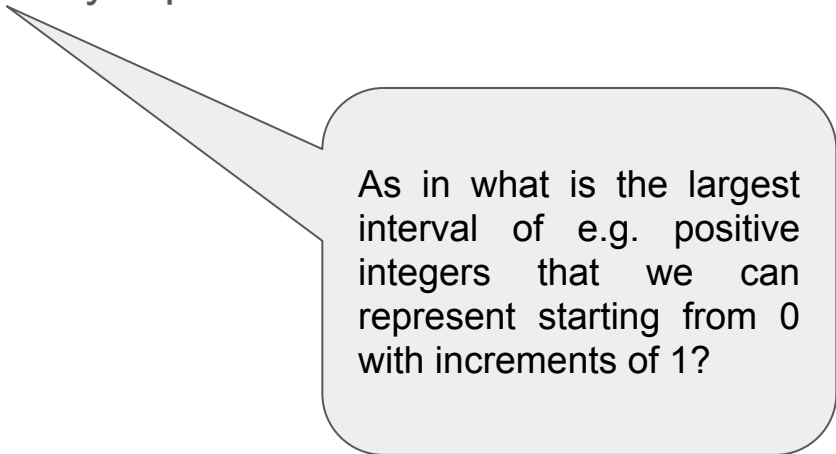
Small floats

<16 bits are not IEEE and seem like a huge sacrifice to make. Still, if you can save on storage you will be asked to do so as long as it does not hurt usability:

Overall bitdepth	Sign bitdepth	Mantissa bitdepth	Exponent bitdepth
16	1	10	5
14**	0*	9	5
11	0*	6	5
10	0	5	5

Floating point formats: IEEE 754

- The higher the magnitude, the smaller the precision
- What is the largest integer it can continuously represent?

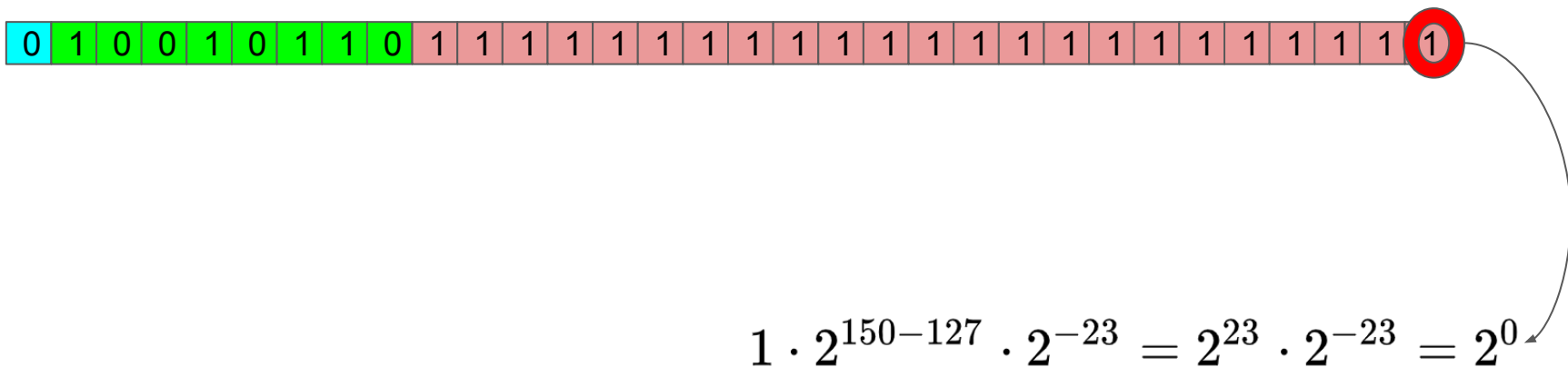


As in what is the largest interval of e.g. positive integers that we can represent starting from 0 with increments of 1?

Floating point formats: IEEE 754

- The higher the magnitude, the smaller the precision
- What is the largest integer it can continuously represent?

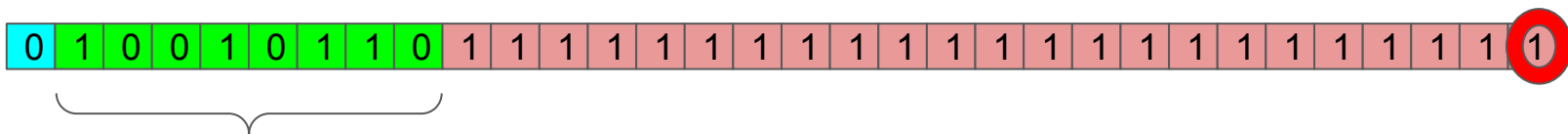
$$s \cdot 2^{r(e)-bias} \cdot (1 + r(m))$$



Floating point formats: IEEE 754

- The higher the magnitude, the smaller the precision
- What is the largest integer it can continuously represent?

$$s \cdot 2^{r(e)-bias} \cdot (1 + r(m))$$



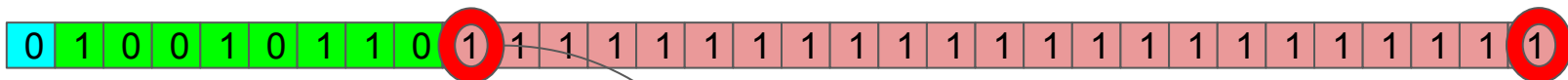
$$r(e) = 2^7 + 2^4 + 2^2 + 2^1 = 150$$

$$1 \cdot 2^{150-127} \cdot 2^{-23} = 2^{23} \cdot 2^{-23} = 2^0$$

Floating point formats: IEEE 754

- The higher the magnitude, the smaller the precision
- What is the largest integer it can continuously represent?

$$s \cdot 2^{r(e)-bias} \cdot (1 + r(m))$$



$$r(e) = 2^7 + 2^4 + 2^2 + 2^1 = 150$$

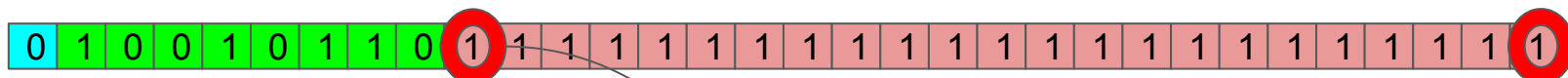
$$1 \cdot 2^{150-127} \cdot 2^{-1} = 2^{23} \cdot 2^{-1} = 2^{22}$$

$$1 \cdot 2^{150-127} \cdot 2^{-23} = 2^{23} \cdot 2^{-23} = 2^0$$

Floating point formats: IEEE 754

- The higher the magnitude, the smaller the precision
- What is the largest integer it can continuously represent?

$$s \cdot 2^{r(e)-bias} \cdot (1 + r(m))$$



$$r(e) = 2^7 + 2^4 + 2^2 + 2^1 = 150$$

$$r(m) = \sum_{i=1}^{23} 2^{-i}$$

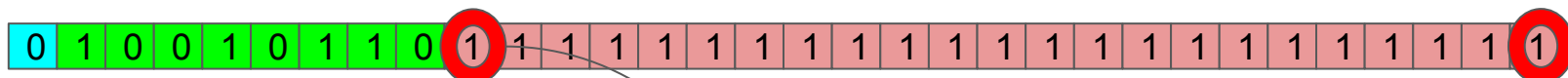
$$1 \cdot 2^{150-127} \cdot 2^{-1} = 2^{23} \cdot 2^{-1} = 2^{22}$$

$$1 \cdot 2^{150-127} \cdot 2^{-23} = 2^{23} \cdot 2^{-23} = 2^0$$

Floating point formats: IEEE 754

- The higher the magnitude, the smaller the precision
- What is the largest integer it can continuously represent?

$$s \cdot 2^{r(e)-bias} \cdot (1 + r(m)) \quad 2^{23} \cdot (1 + \sum_{i=1}^{23} 2^{-i}) = 2^{24} - 1$$



$$r(e) = 2^7 + 2^4 + 2^2 + 2^1 = 150$$

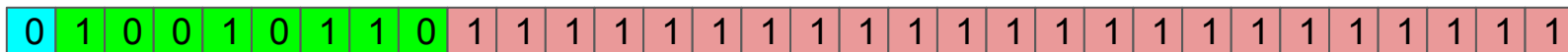
$$r(m) = \sum_{i=1}^{23} 2^{-i}$$

$$1 \cdot 2^{150-127} \cdot 2^{-1} = 2^{23} \cdot 2^{-1} = 2^{22}$$

$$1 \cdot 2^{150-127} \cdot 2^{-23} = 2^{23} \cdot 2^{-23} = 2^0$$

Floating point formats: IEEE 754

- The higher the magnitude, the smaller the precision
- What is the largest integer it can continuously represent?
 - To have all mantissa bits encode integer numbers starting from 1, the biased exponent should be the number of mantissa bits (\Leftrightarrow so we shift all the mantissa bits left). E.g. for float32, the exponent bits should encode 150 because $150 - 127 = 23$
 - For 32 bits, 1 implicit bit in front of the 23 mantissa bits, all set to one = 24 bits; Largest integer is $2^{24}-1$
 - In general: largest integer this way is $2^{(\text{mantissa}+1)}-1$
 - Trick: the largest **consecutive** integer (i.e. without gaps) is $2^{(\text{mantissa}+1)}$ though!
- For binary32 it is $2^{24} = 16'777'216$
- For binary64 it is $2^{53} = 9'007'199'254'740'992$

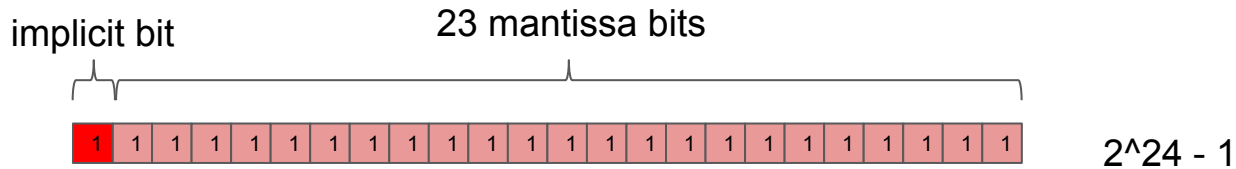


Floating point formats: IEEE 754

What happens when you add $1.0f$ to $2^{24} - 1 = 16'777'215.0f$?

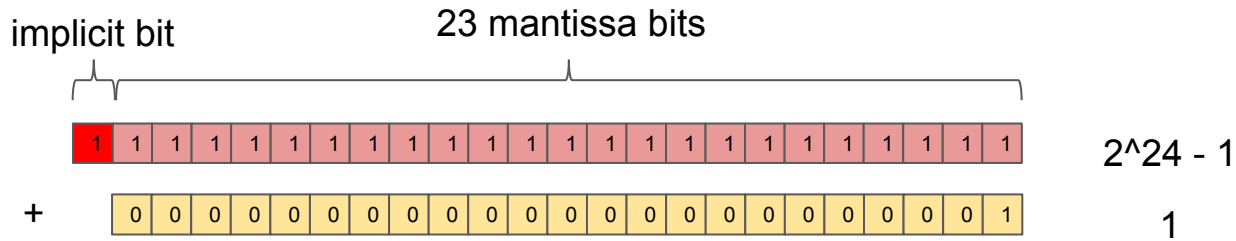
Floating point formats: IEEE 754

What happens when you add 1.0f to $2^{24} - 1 = 16'777'215.0f$?



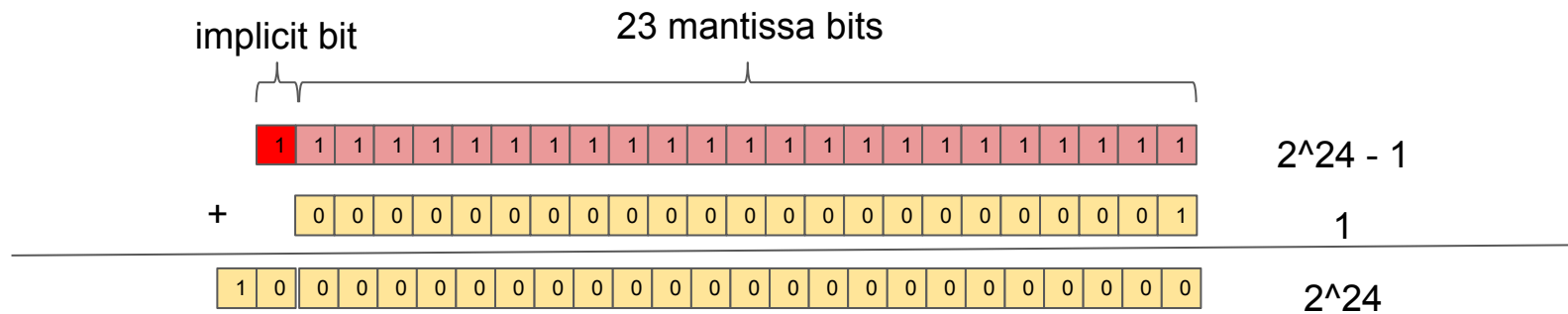
Floating point formats: IEEE 754

What happens when you add 1.0f to $2^{24} - 1 = 16'777'215.0f$?



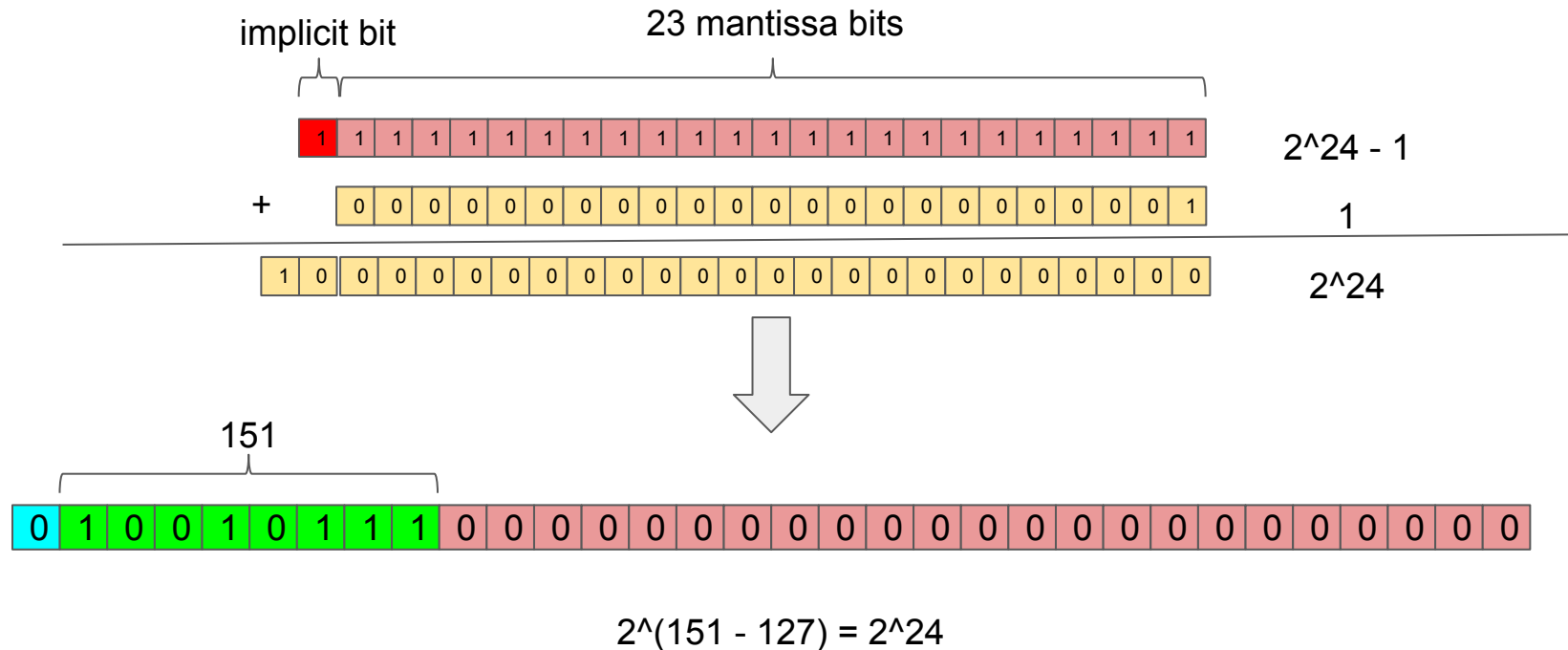
Floating point formats: IEEE 754

What happens when you add 1.0f to $2^{24} - 1 = 16'777'215.0f$?



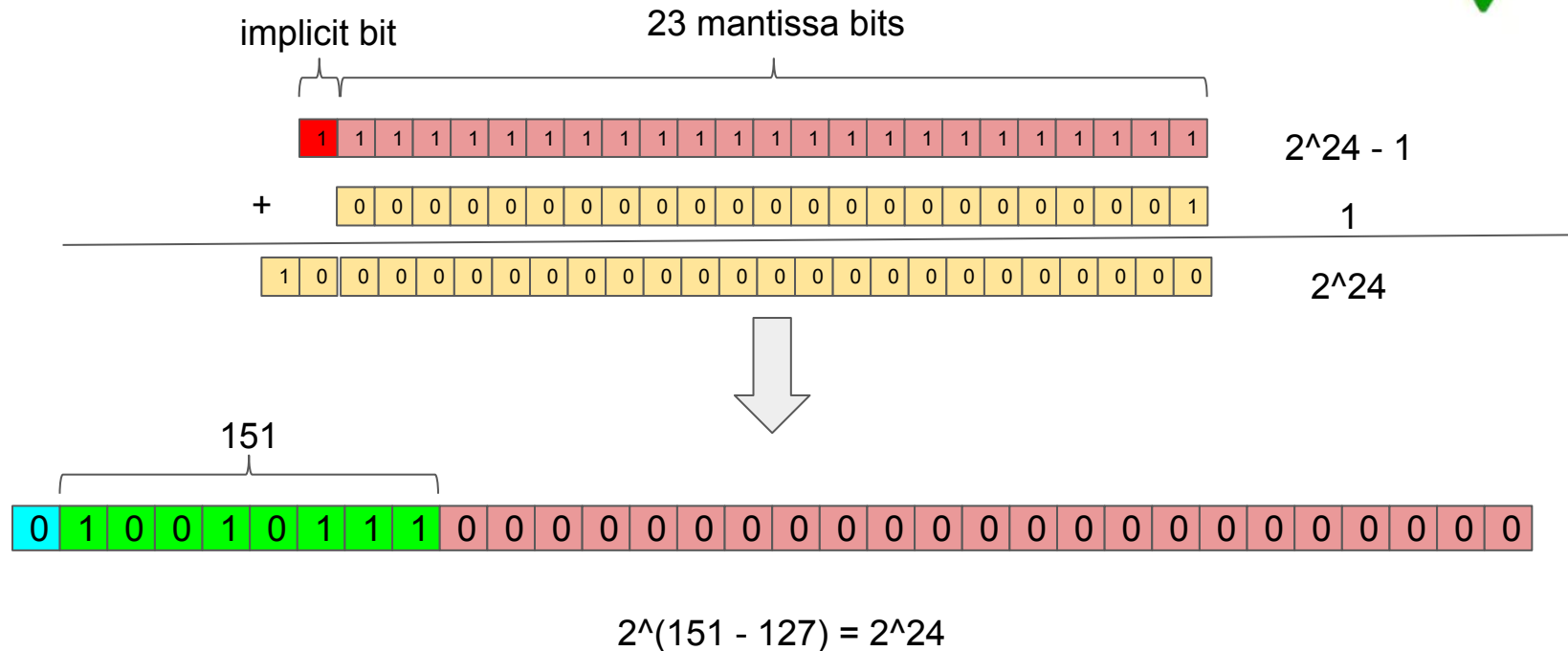
Floating point formats: IEEE 754

What happens when you add 1.0f to $2^{24} - 1 = 16'777'215.0f$?



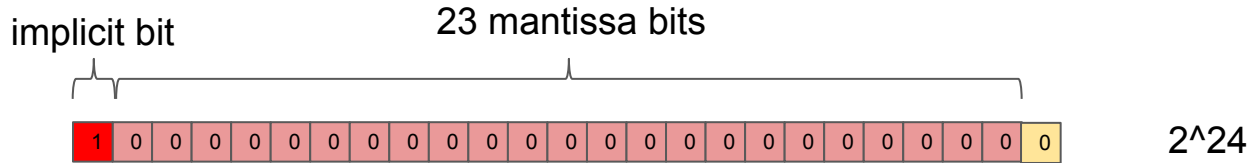
Floating point formats: IEEE 754

What happens when you add 1.0f to $2^{24} - 1 = 16'777'215.0f$?



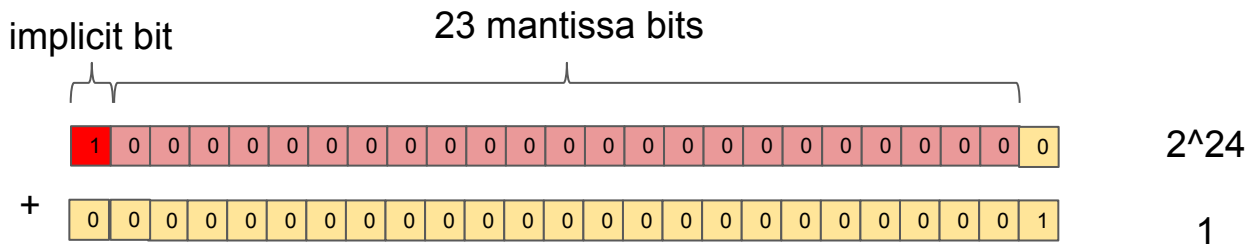
Floating point formats: IEEE 754

What happens when you add $1.0f$ to $2^{24} = 16'777'216.0f$?



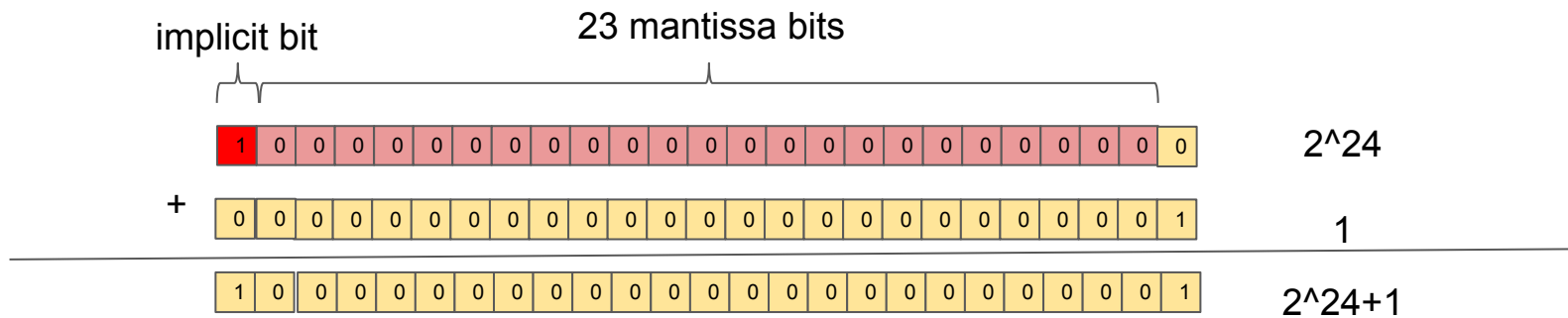
Floating point formats: IEEE 754

What happens when you add 1.0f to $2^{24} = 16'777'216.0f$?



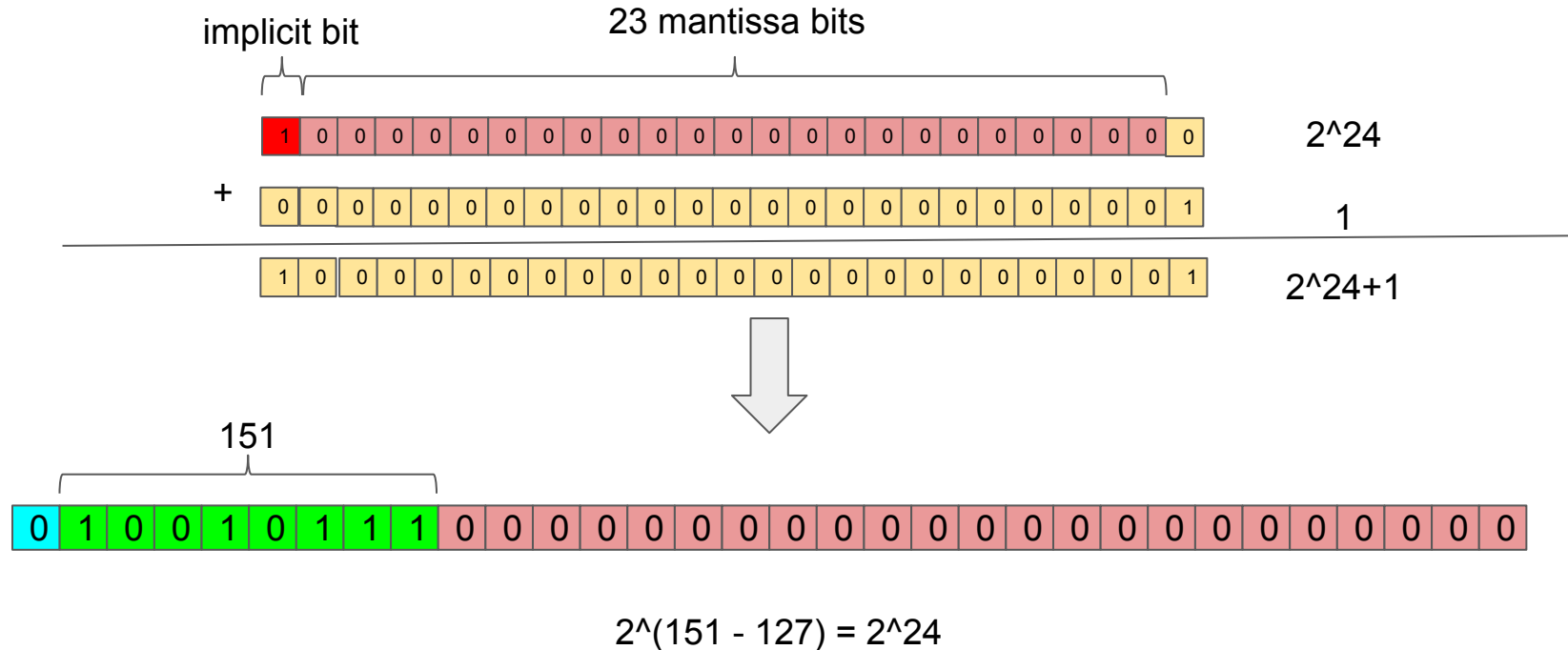
Floating point formats: IEEE 754

What happens when you add $1.0f$ to $2^{24} = 16'777'216.0f$?



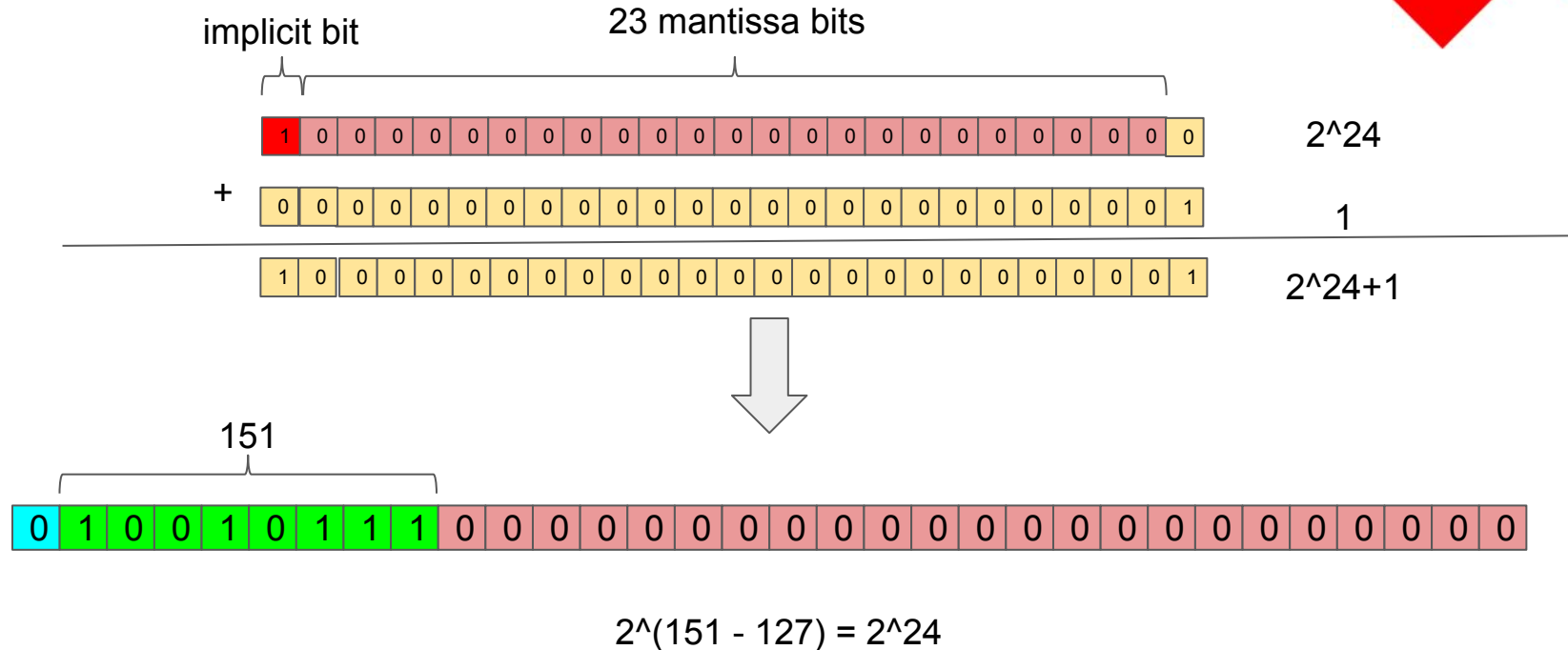
Floating point formats: [IEEE 754](#)

What happens when you add 1.0f to $2^{24} = 16'777'216.0f$?



Floating point formats: IEEE 754

What happens when you add $1.0f$ to $2^{24} = 16'777'216.0f$?

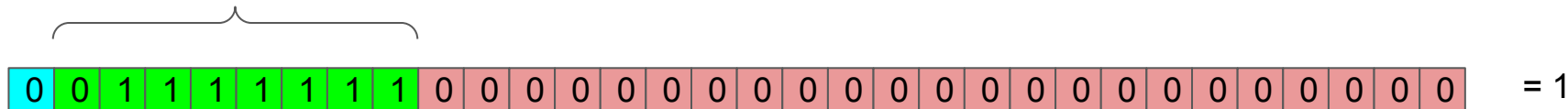


Floating point formats: [IEEE 754](#)

- **Machine epsilon** is the smallest number that satisfies $1.0 + \epsilon > 1.0$

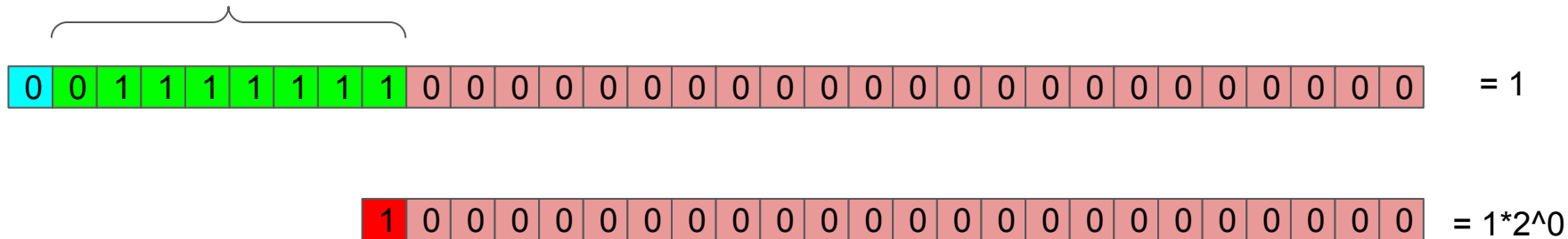
Floating point formats: IEEE 754

- **Machine epsilon** is the smallest number that satisfies $1.0 + \textit{epsilon} > 1.0$

$$127 \Leftrightarrow \text{unbiased} = 127 - 127 = 0$$


Floating point formats: IEEE 754

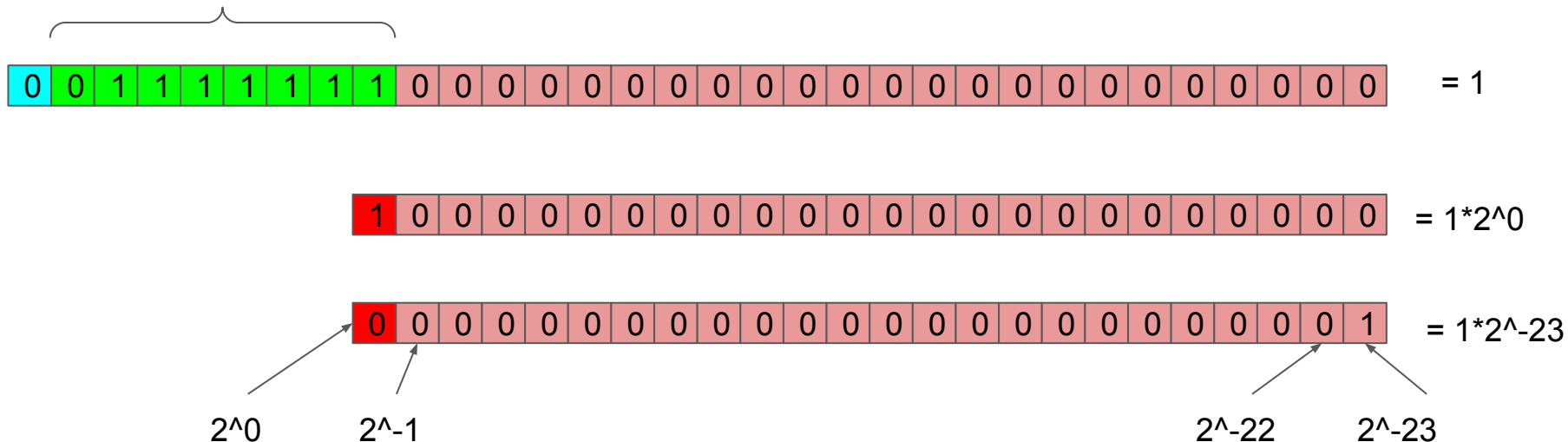
- **Machine epsilon** is the smallest number that satisfies $1.0 + \textit{epsilon} > 1.0$

$$127 \Leftrightarrow \text{unbiased} = 127 - 127 = 0$$


Floating point formats: IEEE 754

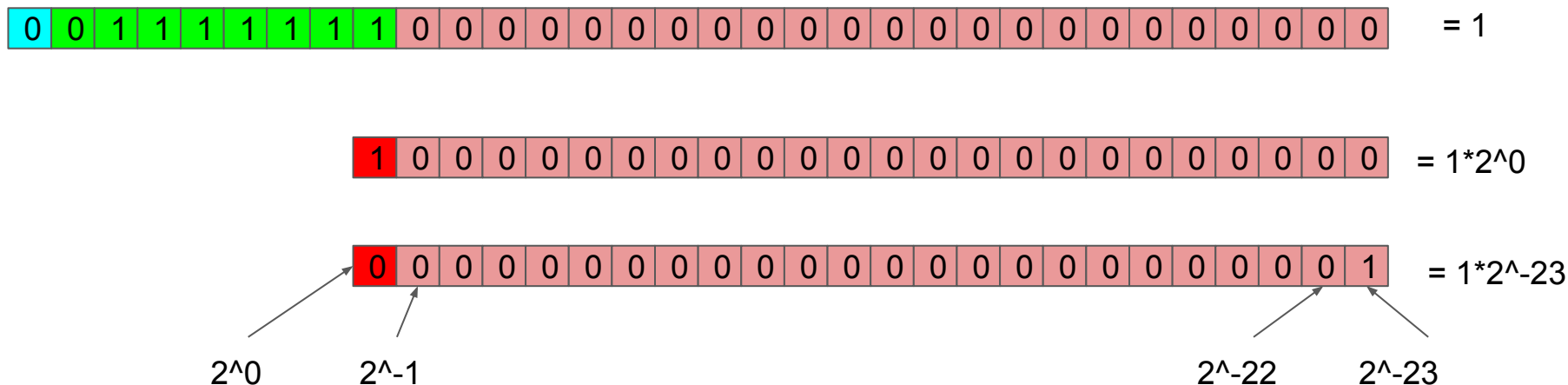
- **Machine epsilon** is the smallest number that satisfies $1.0 + \textit{epsilon} > 1.0$

$127 \Leftrightarrow \text{unbiased} = 127 - 127 = 0$



Floating point formats: IEEE 754

- **Machine epsilon** is the smallest number that satisfies $1.0 + \textit{epsilon} > 1.0$
- We get its value by considering the implicit leading bit 0 and zeroing out the mantissa except for the last bit.
- For 32 bits floats (and truncation) it is $2^{-23} \sim 1.192 \cdot 10^{-7}$

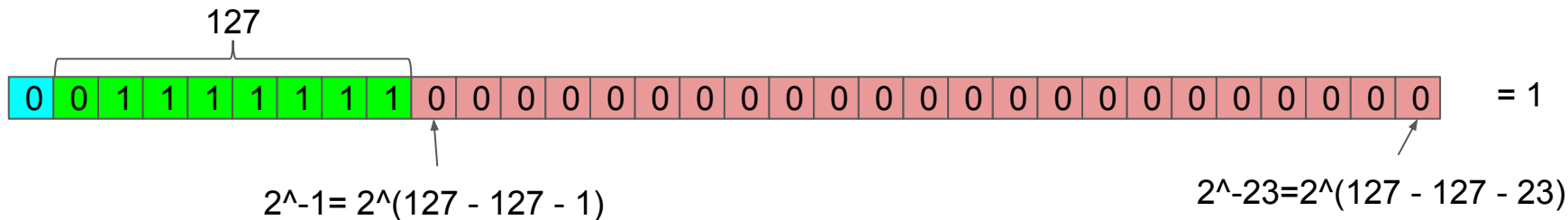


Floating point formats: IEEE 754

- **Units in the last place (ULP)** is the difference between two floating point numbers that coincide in every bit except for the very last mantissa bit

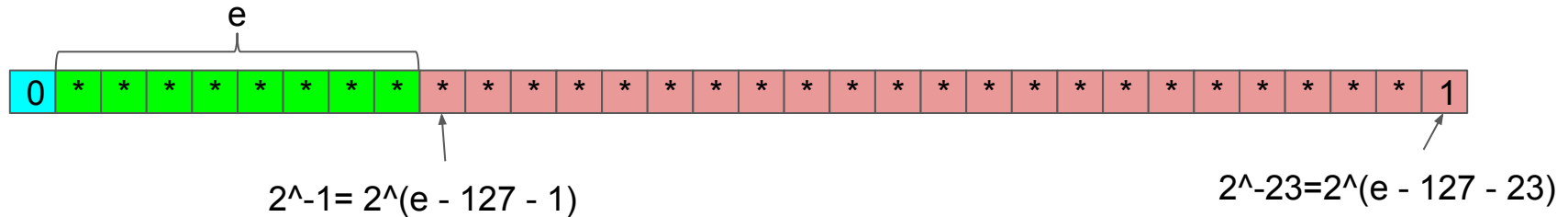
Floating point formats: IEEE 754

- **Units in the last place (ULP)** is the difference between two floating point numbers that coincide in every bit except for the very last mantissa bit



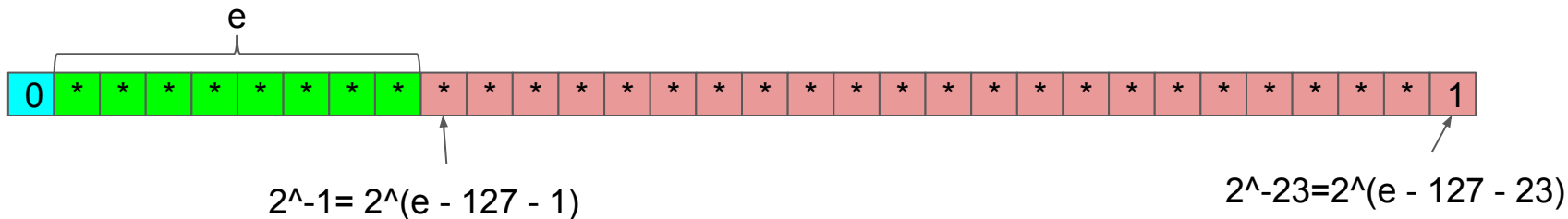
Floating point formats: IEEE 754

- **Units in the last place (ULP)** is the difference between two floating point numbers that coincide in every bit except for the very last mantissa bit

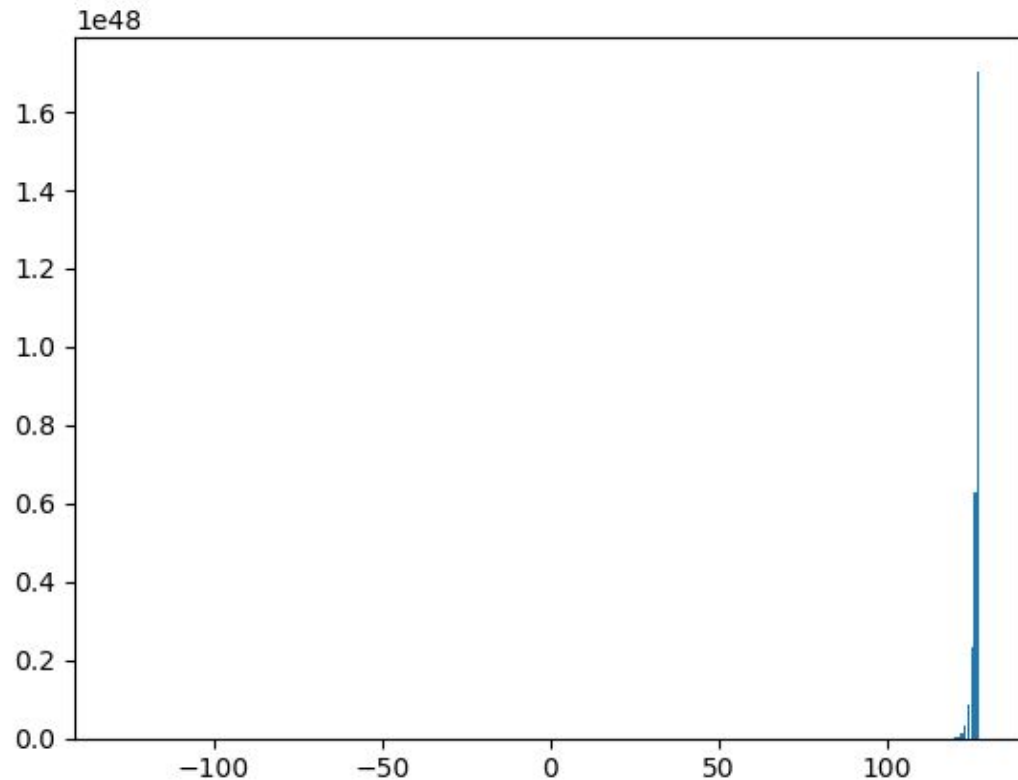


Floating point formats: IEEE 754

- **Units in the last place (ULP)** is the difference between two floating point numbers that coincide in every bit except for the very last mantissa bit
- ULP depends on the exponent; its value is $2^u \cdot \epsilon$, where u is the **unbiased exponent** ($e - 127$, for 32 bit floats) and ϵ is the machine epsilon 2^{-23}



ULP with respect to exponent



ULP with respect to exponent

- **If** $u = -128$ **then** $ULP = 3.06631251889e-63$
- **If** $u = 0$ **then** $ULP = 1.19209289551e-07$
- **If** $u = 127$ **then** $ULP = 1.70494079093e+48$
- If you use a floating point number for accumulation, always compute how long it takes until the increment becomes smaller than ULP
- Actual implementations use more bits for the intermediate results and apply rounding or truncation - so you might get away with e.g. $0.5 * ULP$ if rounding is set to +infinity (or ties away from zero); more on this at https://en.wikipedia.org/wiki/IEEE_754#Rounding_rules
(but in general, prepare for the worse and not the best)

Sidetrack: summing a sequence

- Take 1:

Input: float x[N]

Output: float sum

```
for ( int i=0; i<N; ++i ) sum += x[i];
```

Sidetrack: summing a sequence

- Take 1:

Input: float x[N]

Output: float sum

```
for ( int i=0; i<N; ++i ) sum += x[i];
```

You have to be sure that all $x[i]$ are greater or equal to ULP of the preceding sum. Worst case is when $x[0]$ is so large that adding the rest won't change the sum's value, i.e. $\text{sum} = x[0]$.

Luckily, you can always check if such a problem arises in $\Theta(N)$ time.

Sidetrack from sidetrack

Notation	Name ^[19]	Description	Formal Definition	Limit Definition ^{[20][21][22][19][14]}
$f(n) = o(g(n))$	Small O; Small Oh	f is dominated by g asymptotically	$\forall k > 0 \exists n_0 \forall n > n_0 f(n) < k \cdot g(n)$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
$f(n) = O(g(n))$	Big O; Big Oh; Big Omicron	$ f $ is bounded above by g (up to constant factor) asymptotically	$\exists k > 0 \exists n_0 \forall n > n_0 f(n) \leq k \cdot g(n)$	$\limsup_{n \rightarrow \infty} \frac{ f(n) }{g(n)} < \infty$
$f(n) = \Theta(g(n))$	Big Theta	f is bounded both above and below by g asymptotically	$\exists k_1 > 0 \exists k_2 > 0 \exists n_0 \forall n > n_0$ $k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$	$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ (Knuth version)
$f(n) \sim g(n)$	On the order of	f is equal to g asymptotically	$\forall \varepsilon > 0 \exists n_0 \forall n > n_0 \left \frac{f(n)}{g(n)} - 1 \right < \varepsilon$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$
$f(n) = \Omega(g(n))$	Big Omega in number theory (Hardy–Littlewood)	$ f $ is not dominated by g asymptotically	$\exists k > 0 \forall n_0 \exists n > n_0 f(n) \geq k \cdot g(n)$	$\limsup_{n \rightarrow \infty} \left \frac{f(n)}{g(n)} \right > 0$
$f(n) = \Omega(g(n))$	Big Omega in complexity theory (Knuth)	f is bounded below by g asymptotically	$\exists k > 0 \exists n_0 \forall n > n_0 f(n) \geq k \cdot g(n)$	$\liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$
$f(n) = \omega(g(n))$	Small Omega	f dominates g asymptotically	$\forall k > 0 \exists n_0 \forall n > n_0 f(n) > k \cdot g(n) $	$\lim_{n \rightarrow \infty} \left \frac{f(n)}{g(n)} \right = \infty$

People almost always mean Theta or ‘in the order of’ instead of Ordo when they say something like ‘this algorithm is $O(\dots)$ ’. Linear search is $O(n)$ and also $O(n^2)$, etc. The only Theta it belongs to is $\Theta(N)$.

Sidetrack: summing a sequence

- Take 2: sort the array before summing

Input: `float x[N]`

Output: `float sum`

```
sort( x, ASCENDING );
```

```
for ( int i=0; i<N; ++i ) sum += x[i];
```

Sidetrack: summing a sequence

- Take 2: sort the array before summing

Input: `float x[N]`

Output: `float sum`

```
sort( x, ASCENDING );
```

```
for ( int i=0; i<N; ++i ) sum += x[i];
```

You can still get into trouble because of accumulating rounding errors. For a solution, see https://en.wikipedia.org/wiki/Kahan_summation_algorithm

Sidetrack from sidetrack from sidetrack

```
function KahanSum(input)
    var sum = 0.0
    var c = 0.0

    for i = 1 to input.length do
        var y = input[i] - c // c is zero the first time around.
        var t = sum + y      // low-order digits of y are lost.
        c = (t - sum) - y   // (t - sum) cancels the high-order
                           // part of y; subtracting y recovers
                           // negative (low part of y)
        sum = t             // unsafe opt. warning!
    next i
    return sum
```

https://en.wikipedia.org/wiki/Kahan_summation_algorithm

Subnormals (or denormals)

- There's a gap around zero: no 32 bit floating point numbers lie in the $(-2^{-126}, +2^{-126}) \sim (-1.175 \cdot 10^{-38}, +1.175 \cdot 10^{-38})$ interval
- **Denormals** (or **subnormals**) fill this gap
- Denormals are represented by their unbiased exponent at 0 (so with a biased exponent of $0-127=-127$ in case of binary32), but their implicit leading 1 bit is treated as a 0
- So denormals fill up the $(-2^{-126}, +2^{-126})$ range evenly
- As such, the smallest possible floating point number with denormals is

$$2^{-126} \times 2^{-23} = 2^{-149} \approx 1.4012984643 \times 10^{-45}$$

Subnormals (or denormals)

- Check your architecture: subnormals can mean a prohibitive performance hit
- Many high performance solutions simply flush them to zero
- E.g. in audio usually a denormal would mean an inaudible signal
- You can flush them to zero on the GPU too (e.g.

<https://devblogs.nvidia.com/cuda-pro-tip-flush-denormals-confidence/>)

Floating point numbers



<https://www.volkerschatz.com/science/float.html>

Inf and NaN (for binary32)

- The highest possible exponent represents infinity and not-a-numbers (NaNs)
- **If** $e = 127$ **and** mantissa = 0 **then** +/-infinity
- **If** $e = 127$ **and** mantissa $\neq 0$ **then** NaN
- Special rules apply to them
 - 'The comparisons EQ, GT, GE, LT, and LE, when either or both operands is NaN returns FALSE.'
 - 'The comparison NE, when either or both operands is NaN returns TRUE.'
 - +/- infinity in relation to normal numbers behave as you'd expect (they bound them)
- Check if your compiler is IEEE 754 compliant, in C++ with `<limits>`'s `numeric_limits::is_iec559` (IEC559 ~ IEEE 754)
- Even if they are, it can bite you. Very bad.
- GPU architectures and APIs are a different story - always check these! Not even [intrinsics](#) are trivial (*unless you are already used to it but in that case there's precious little for you in this presentation*).

NaNs

- Two kinds: signalling and quiet NaNs
- NaNs are produced by:
 - $\infty - \infty$,
 - $-\infty + \infty$,
 - $0 \times \infty$,
 - $0 \div 0$,
 - $\infty \div \infty$

Operations involving infs and NaNs

$x * \text{INF} \rightarrow \text{INF}$ for $x > 0$

$x * -\text{INF} \rightarrow \text{INF}$ for $x < 0$

$\text{INF} - \text{INF} \rightarrow \text{NaN}$

$x / 0 \rightarrow \text{INF}$ for $x > 0$

$0 * \text{INF} \rightarrow \text{NaN}$

$0 / 0 \rightarrow \text{NaN}$

$x * \text{NaN} \rightarrow \text{NaN}$

Out of range function argument (e.g. `sqrt(-1)`) $\rightarrow \text{NaN}$

Where x is either a regular number or +/- infinity.

IEEE binary32 floating point numbers

if $e = 127$ **and** mantissa = 0 **then** +/-infinity
else if $e = 127$ **and** mantissa $\neq 0$ **then** NaN
else if $e = -127$ **then** denormal
else regular floating point number

IEEE binary32 floating point numbers

if $e = 127$ **and** mantissa = 0 **then** +/-infinity
else if $e = 127$ **and** mantissa $\neq 0$ **then** NaN
else if $e = -127$ **then** denormal
else regular floating point number

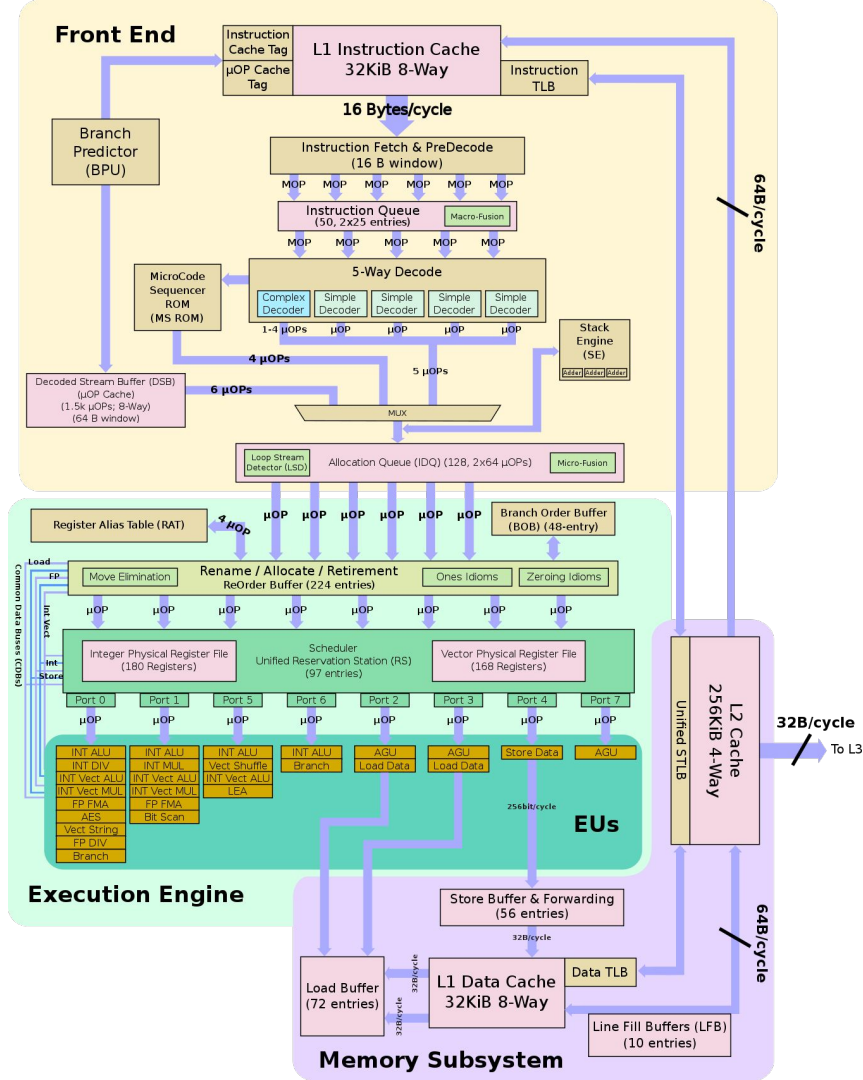
<https://www.amazon.com/Handbook-Floating-Point-Arithmetic-Jean-Michel-Muller/dp/081764704X>

Floating point operations

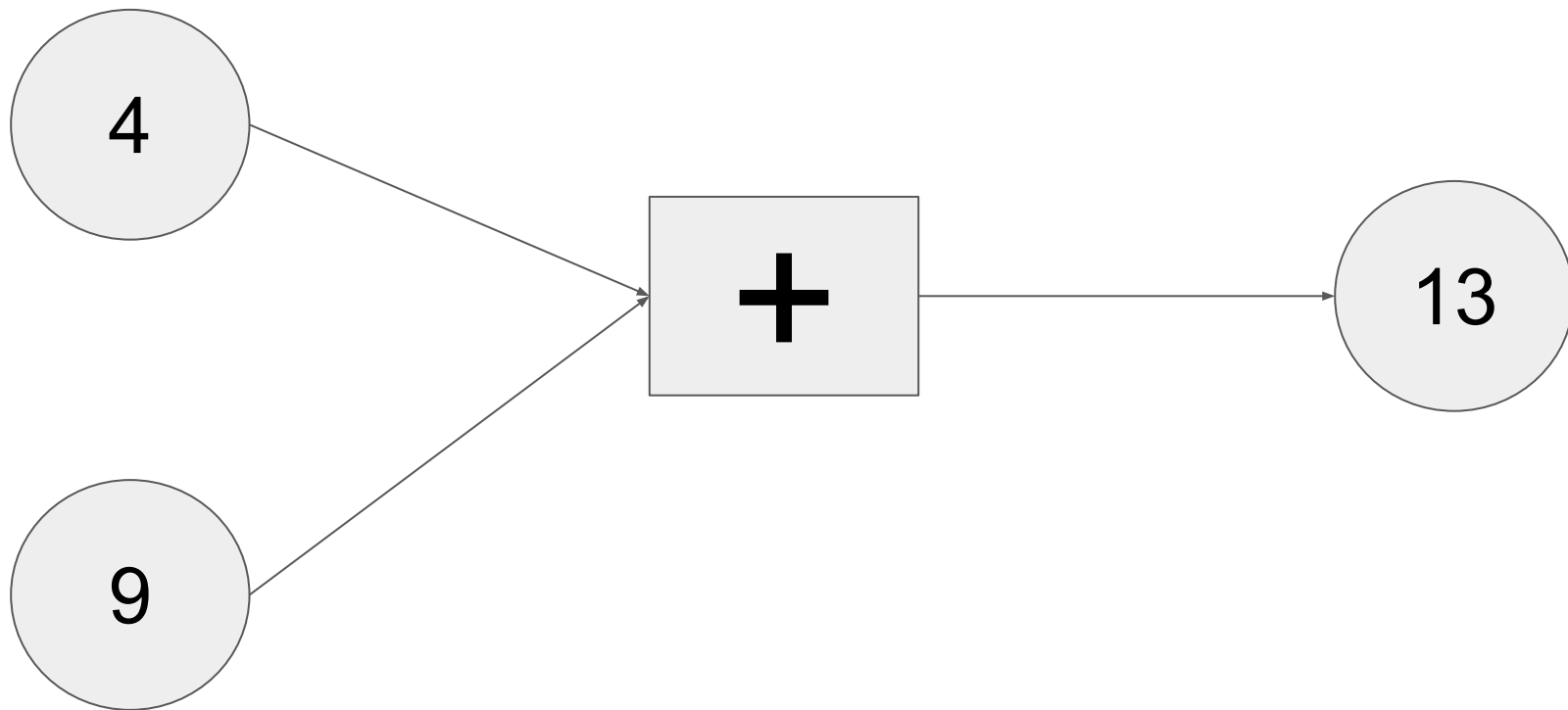
- **Multiplication:** mostly safe, but not always what you expect (i.e. $0 * x$ is not always zero: if $x = \text{NaN}$, then $0 * \text{NaN} = \text{NaN}$)
- **Division:**
 - Check if the denominator is zero
 - Handle the $\text{denom}=0$ edge case (compute expression's limit, etc.)
 - Sometimes this means a more elaborate change: see $\sin(x) / x$
- **Addition, subtraction:** smaller magnitude number can disappear
- **Subtracting** two nearly equal numbers may result in [catastrophic cancellation](#)
- And these are assuming you are in IEEE compliant mode. E.g. some GPU non-IEEE modes treat $0 * x = 0$, i.e. even NaN can be zeroed out. And if you think desktop GPU programming has idiosyncrasies, try WebGL...

Actual CPU architectures

https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake#Individual_Core

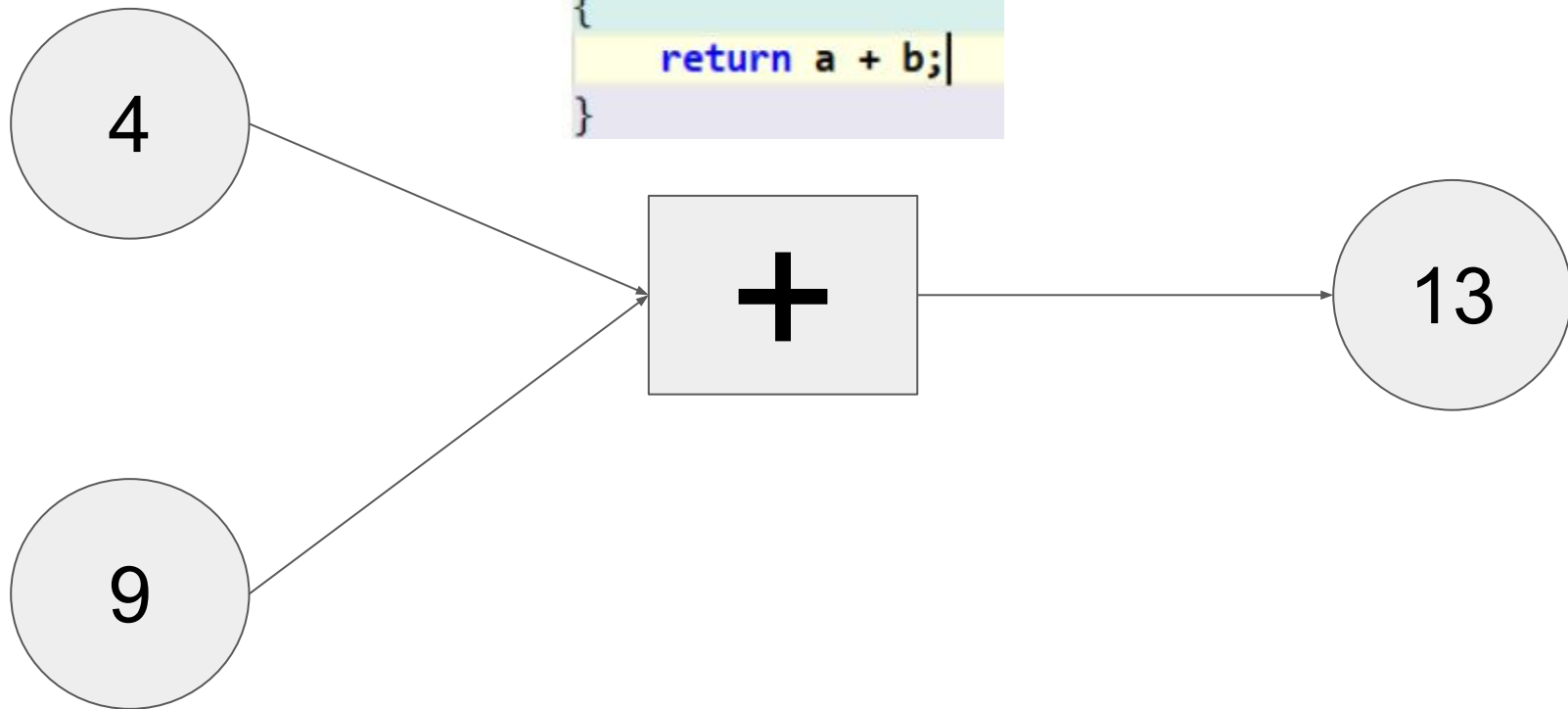


Simple function



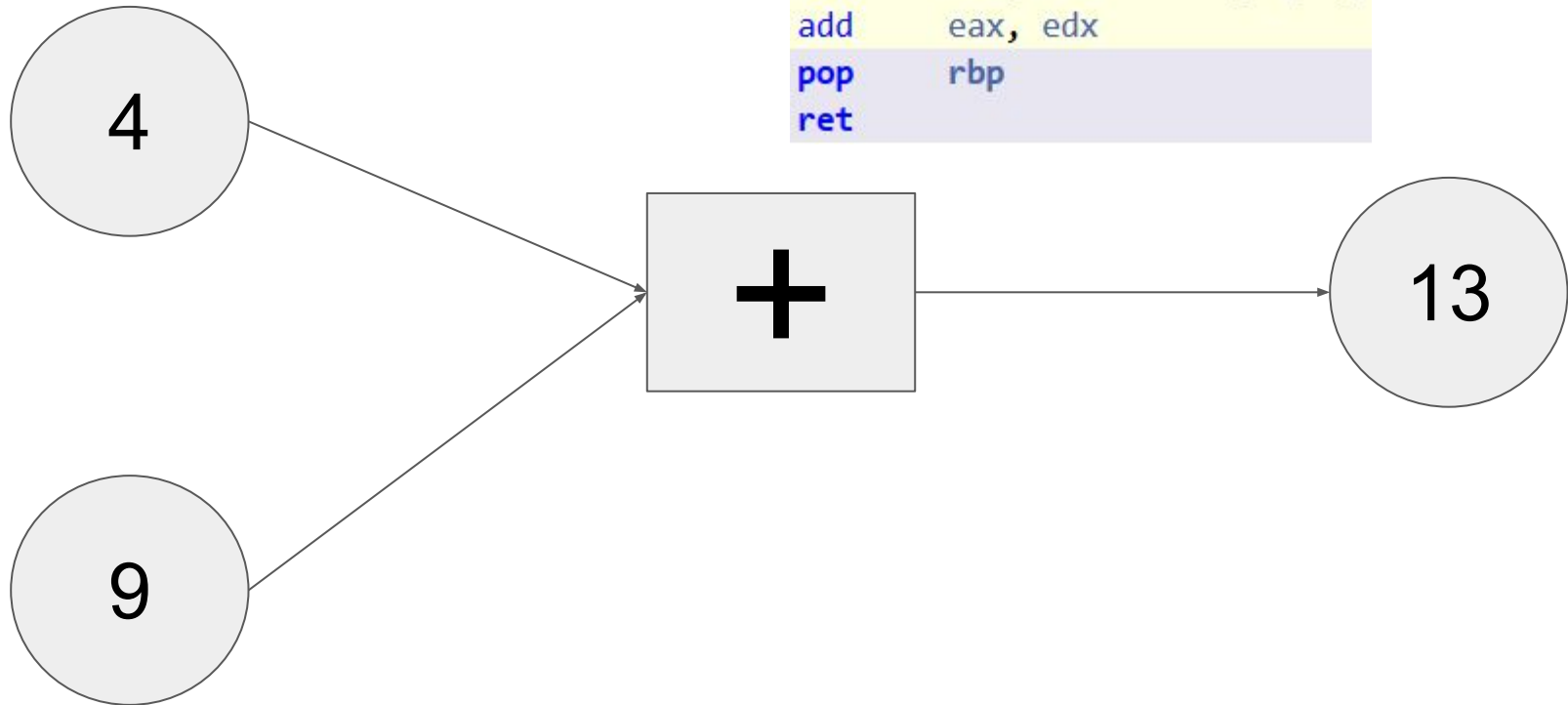
Simple function

```
int f(int a, int b)
{
    return a + b;
}
```



Simple function

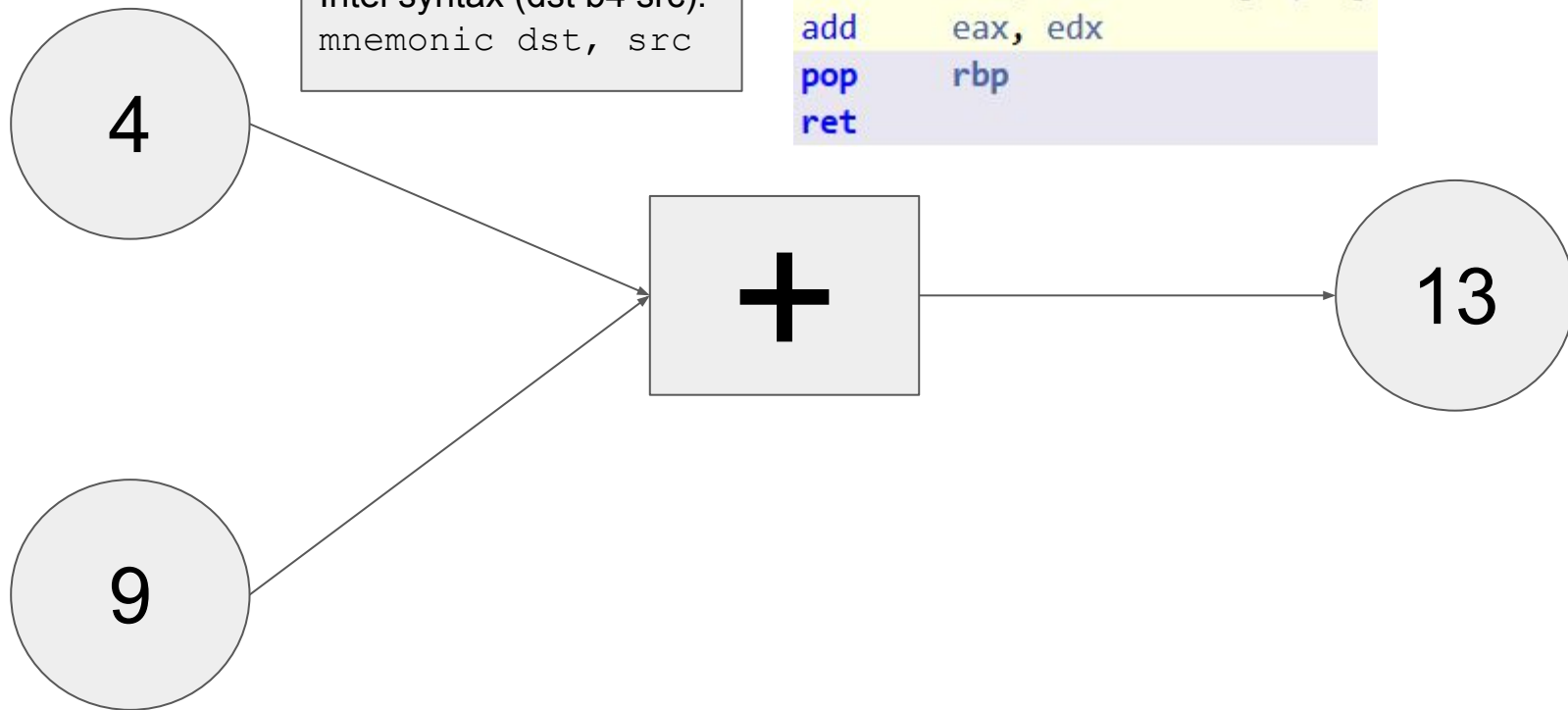
```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-4], edi
mov     DWORD PTR [rbp-8], esi
mov     edx, DWORD PTR [rbp-4]
mov     eax, DWORD PTR [rbp-8]
add     eax, edx
pop     rbp
ret
```



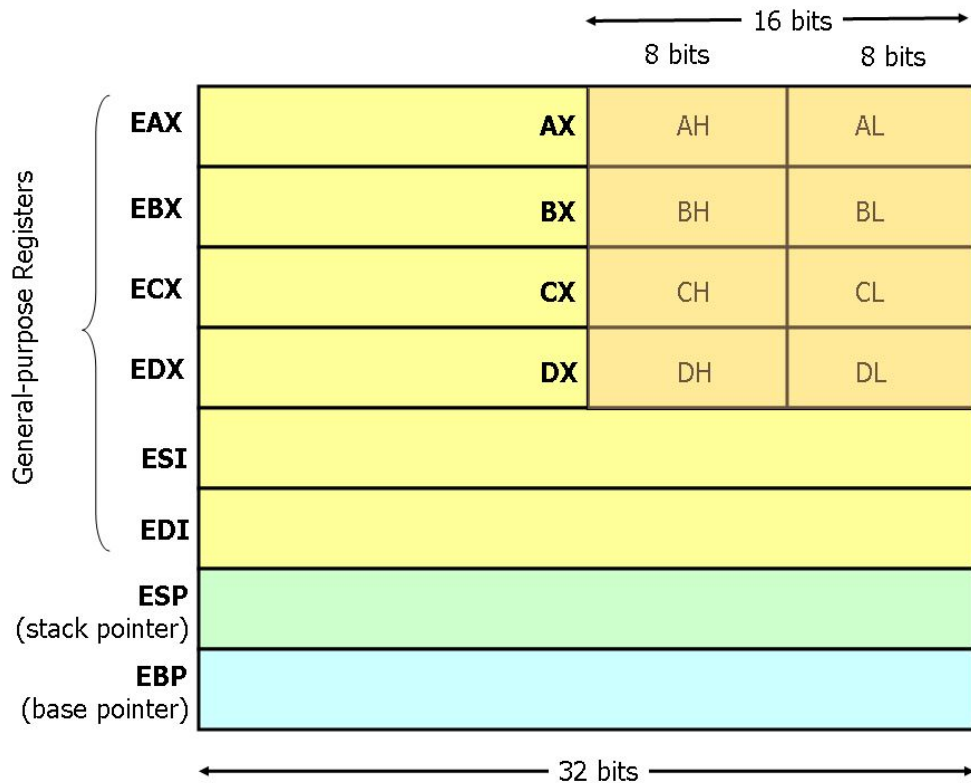
Simple function

Intel syntax (dst b4 src):
mnemonic dst, src

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-4], edi
mov     DWORD PTR [rbp-8], esi
mov     edx, DWORD PTR [rbp-4]
mov     eax, DWORD PTR [rbp-8]
add     eax, edx
pop     rbp
ret
```



Registers



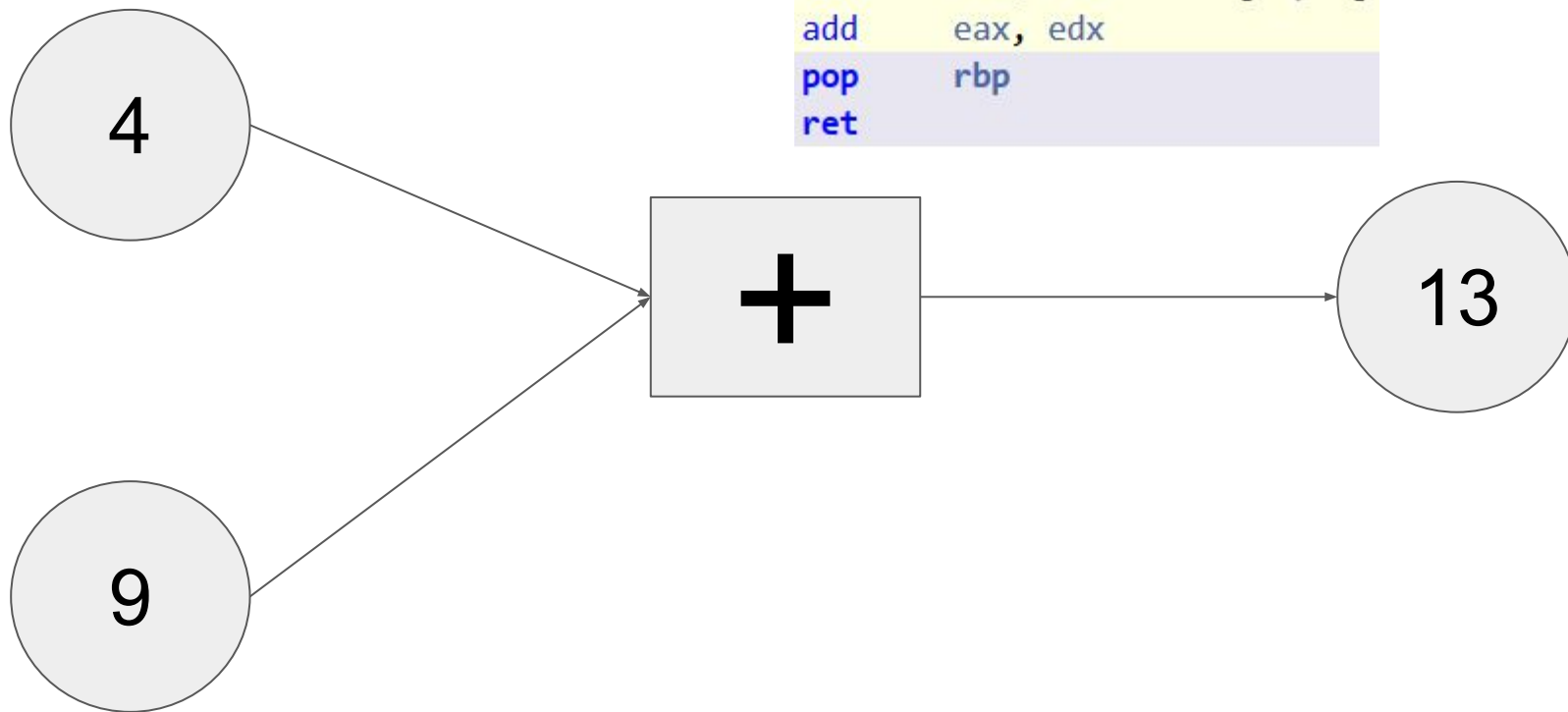
Commands

Three main categories:

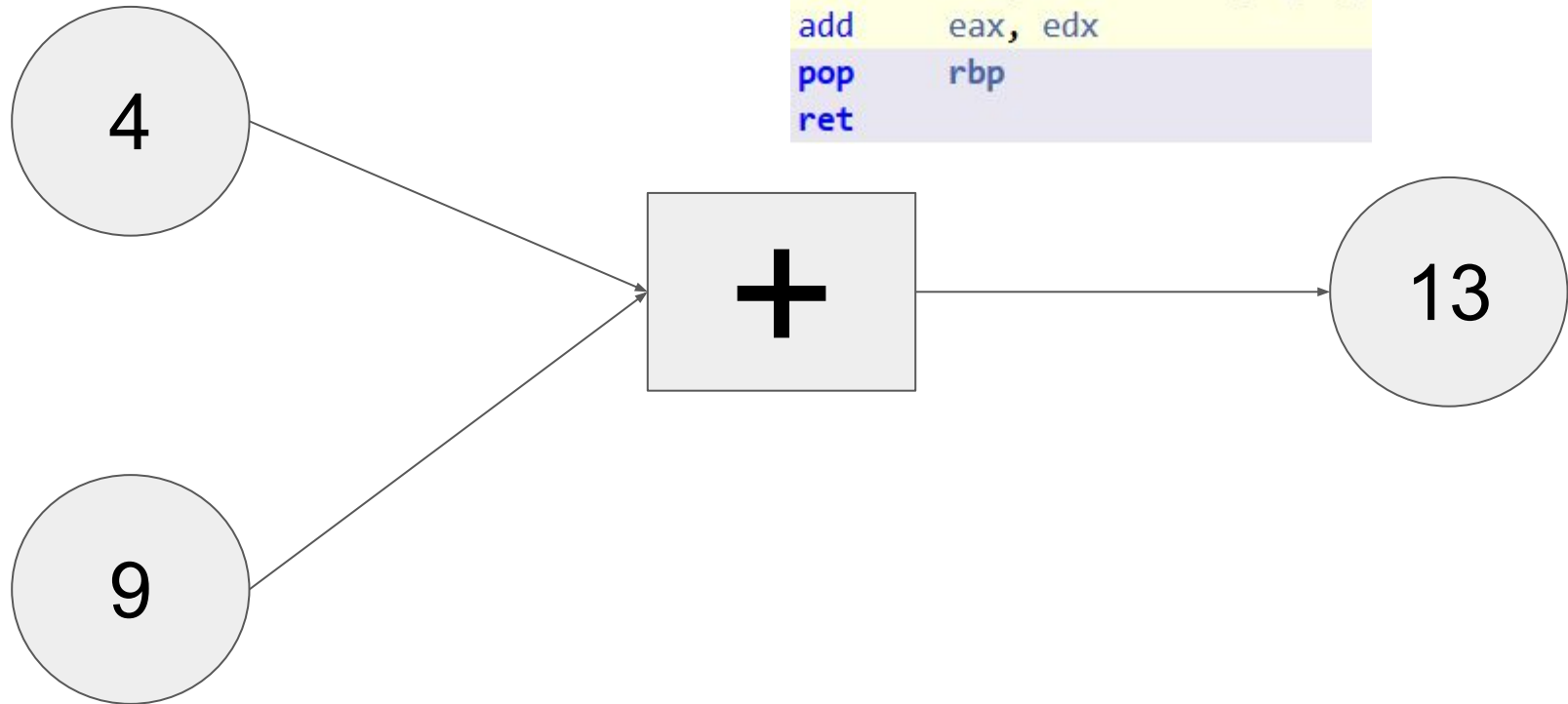
- **Data movement:** `mov, push, pop, etc.`
- **Arithmetic/logic operations:** `add, sub, imul, mul, div, and, xor, etc.`
- **Control flow:** `jmp, je, jne, call, ret, etc.`

Simple function

```
push    rbp
mov     rbp, rsp
{mov     DWORD PTR [rbp-4], edi
mov     DWORD PTR [rbp-8], esi
mov     edx, DWORD PTR [rbp-4]
mov     eax, DWORD PTR [rbp-8]
add     eax, edx
pop     rbp
ret}
```



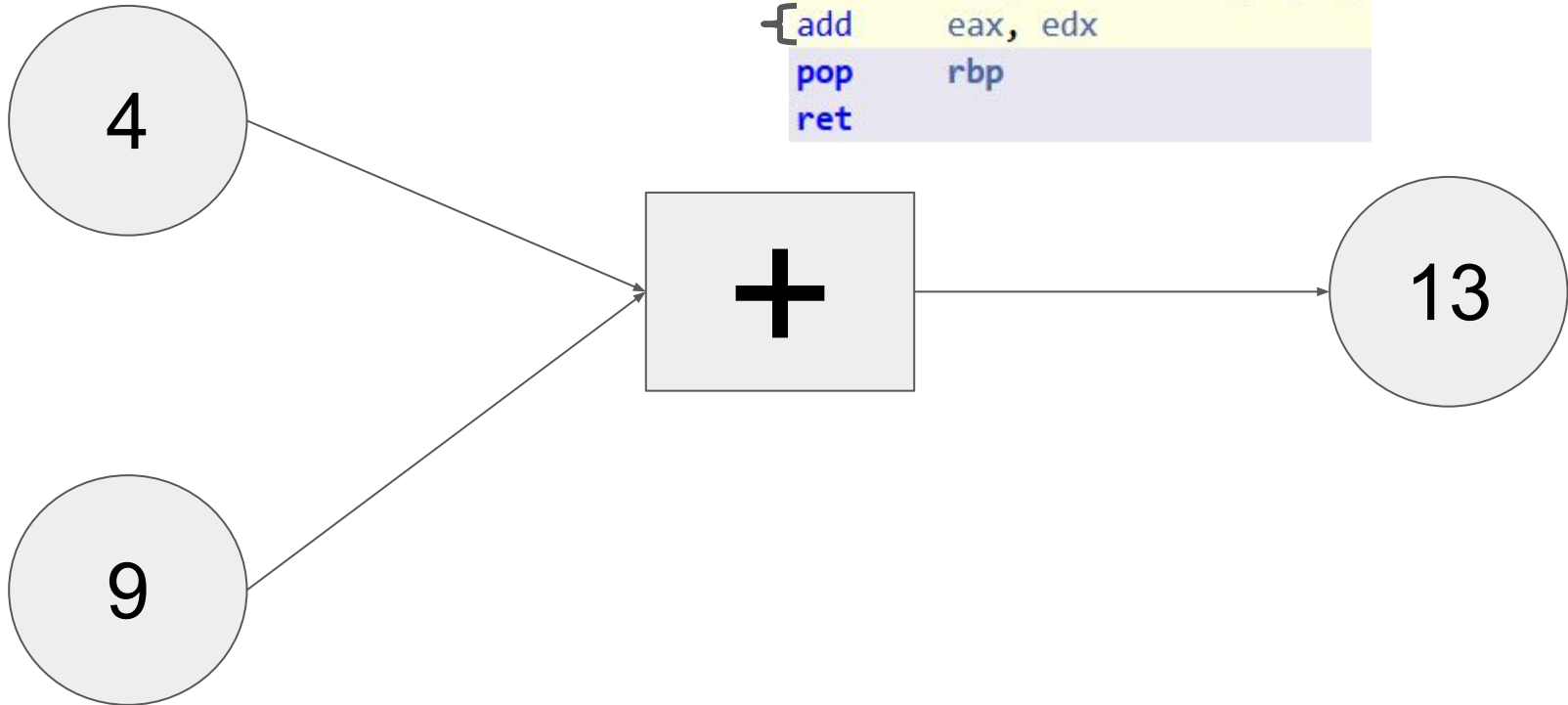
Simple function



```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-4], edi
mov     DWORD PTR [rbp-8], esi
{mov     edx, DWORD PTR [rbp-4]
mov     eax, DWORD PTR [rbp-8]
add     eax, edx
pop     rbp
ret
```

Simple function

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-4], edi
mov     DWORD PTR [rbp-8], esi
mov     edx, DWORD PTR [rbp-4]
mov     eax, DWORD PTR [rbp-8]
add     eax, edx
pop     rbp
ret
```



Architectures: pipelining, superscalars

Pipeline architectures

Fetch: fetch the next instruction pointed by the program counter (PC). Fetch predictor sets the PC to the next predicted command of the program.

fetch

decode

execute

memory

register
write-back

Pipeline architectures

Decode Stage: interpret the instruction. Identify the named registers and read their values from the register file.

fetch

decode

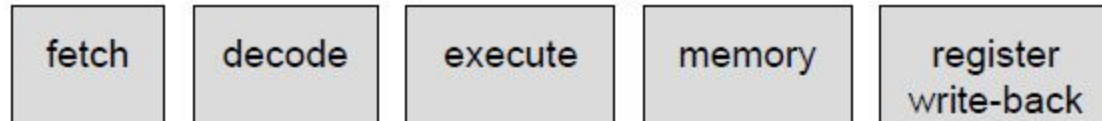
execute

memory

register
write-back

Pipeline architectures

Execute Stage: The control unit of the CPU passes the decoded information as a sequence of control signals to the relevant function units of the CPU to perform the actions required by the instruction, such as reading values from registers, passing them to the ALU to perform mathematical or logic functions on them, and writing the result back to a register. If the ALU is involved, it sends a condition signal back to the CU (control unit). The result generated by the operation is stored in the main memory or sent to an output device. Based on the feedback from the ALU, the PC may be updated to a different address from which the next instruction will be fetched.



Pipeline architectures

Memory access: access data memory.

fetch

decode

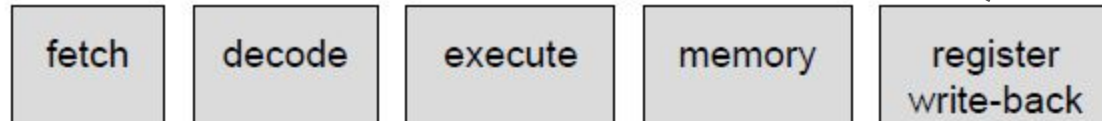
execute

memory

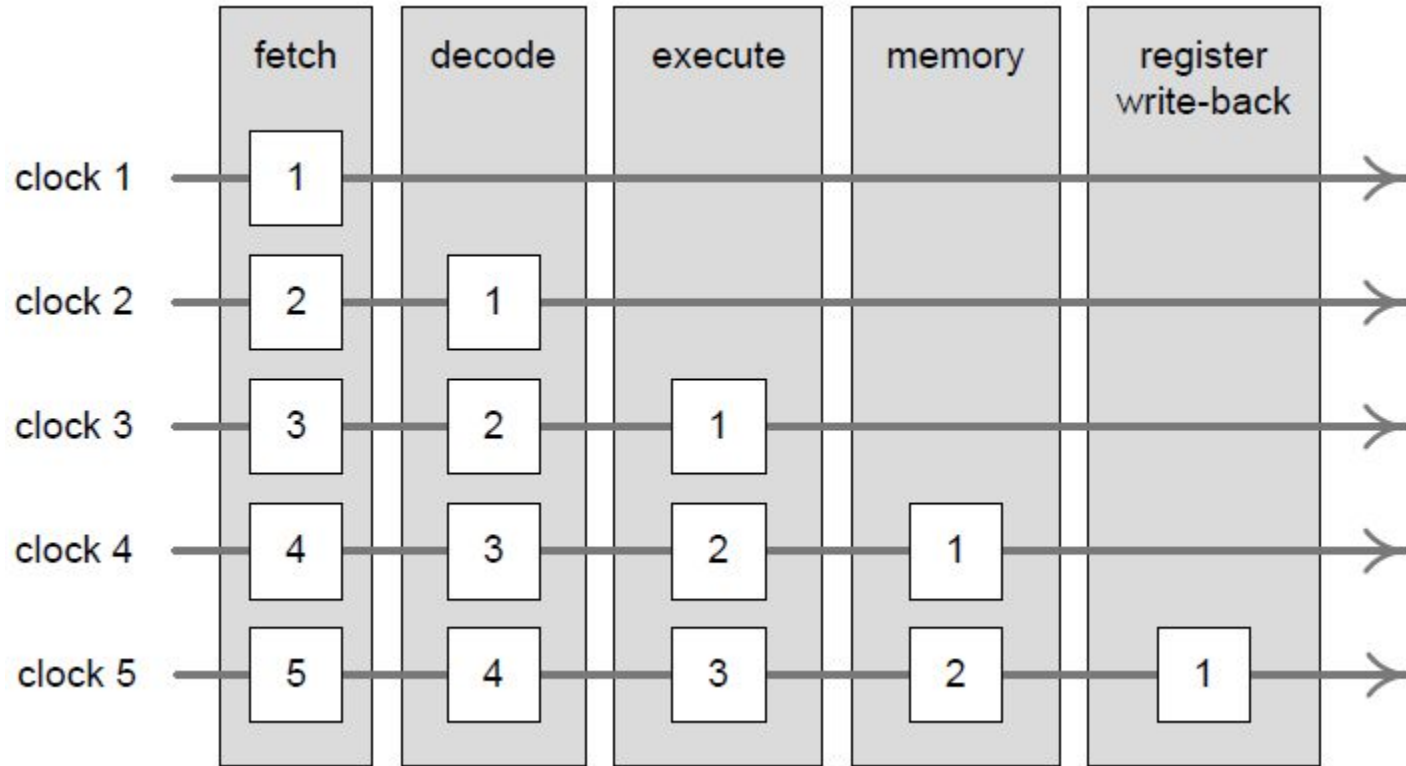
register
write-back

Pipeline architectures

Write-back: write back results to register file.



Pipeline architectures

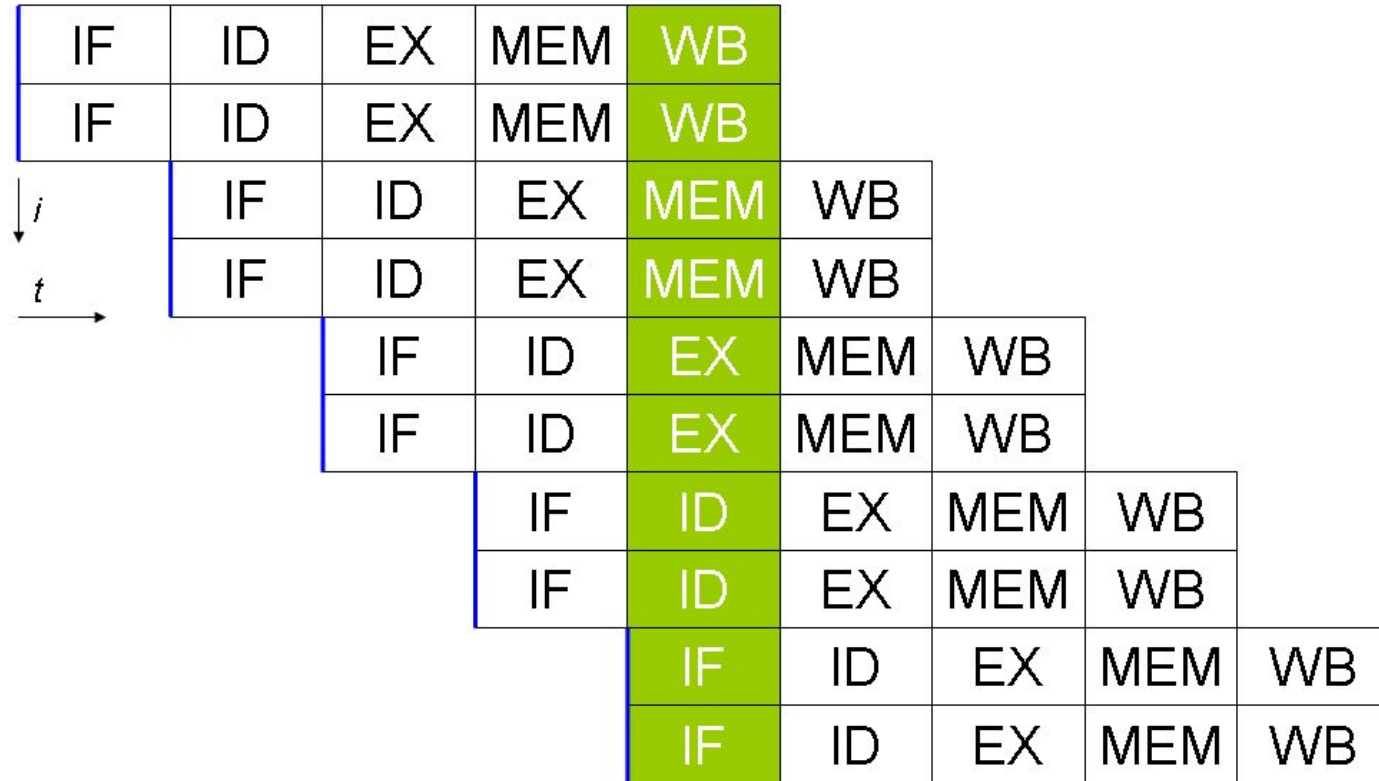


Pipeline architectures

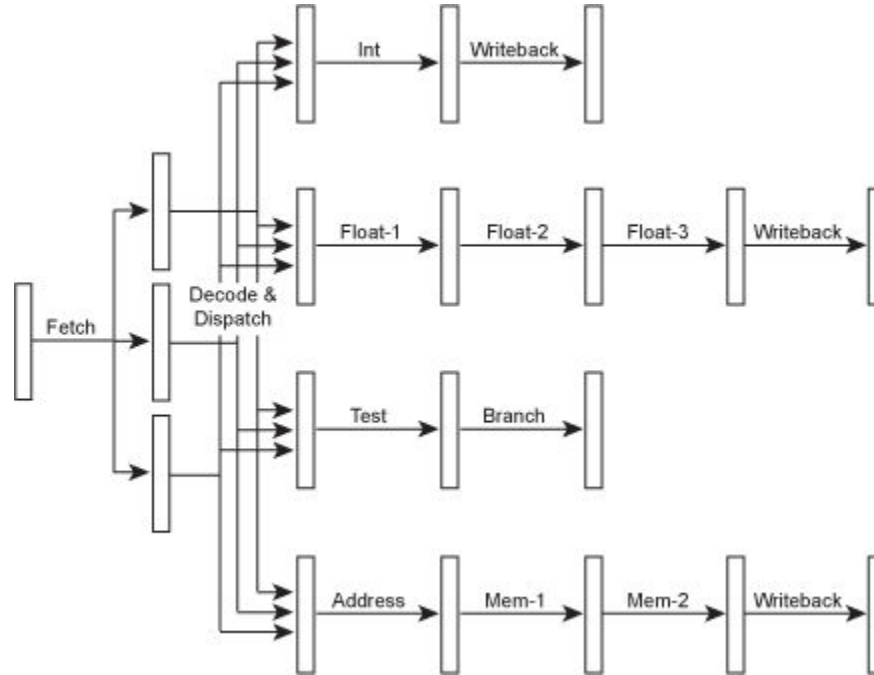
Two useful statistics:

- **Latency**: how many cycles it takes to finish an instruction
- **Bandwidth** or **throughput**: how many instructions are in-flight per unit time.

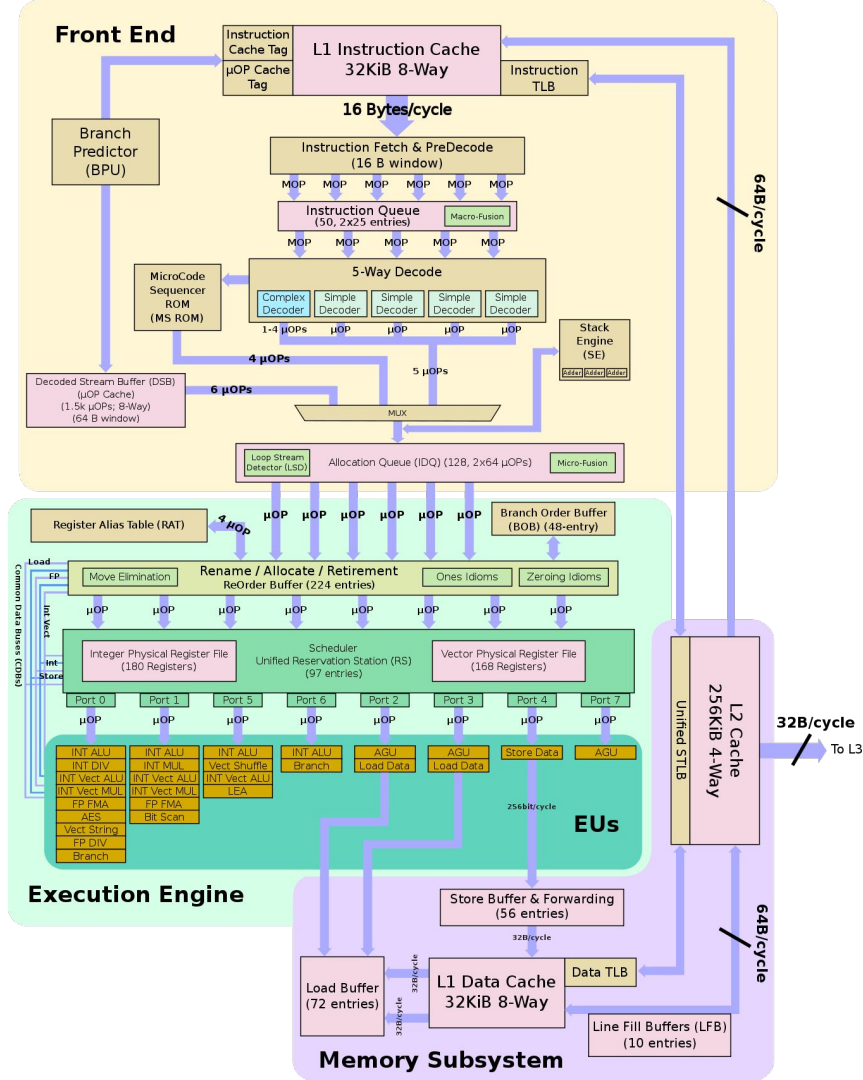
Superscalar architectures



Superscalar architectures



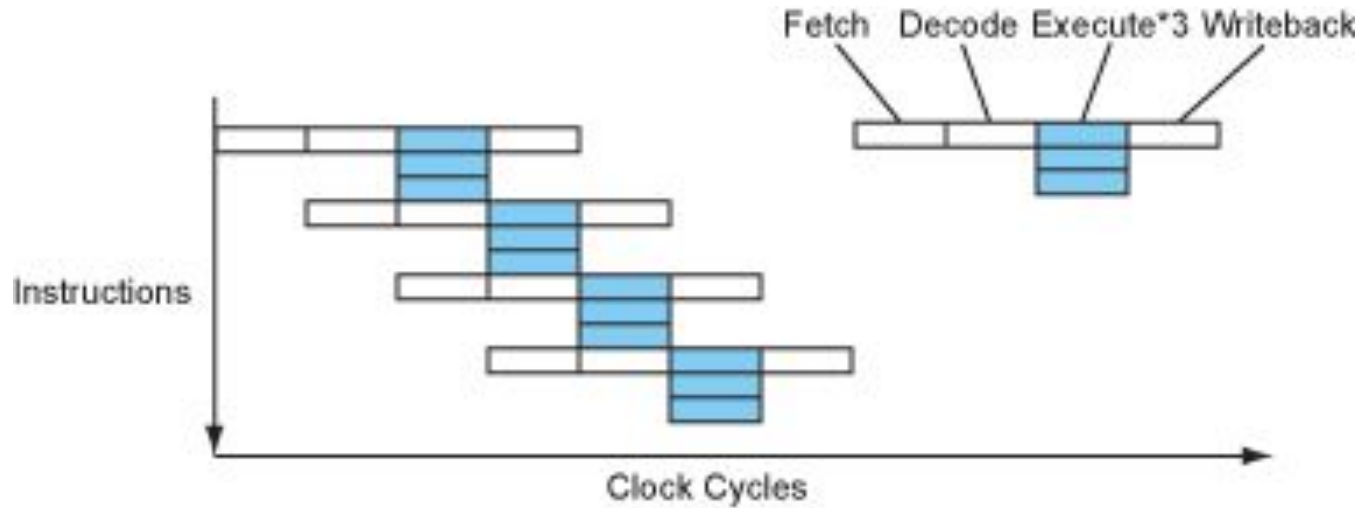
https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake#Individual_Core



Superscalar architectures

- Multiple redundant units to implement various stages of the pipeline
- As such, multiple instructions can be executed concurrently
- Modern processors are pipelined, superscalar architectures
- More on these:
 - <http://www.lighterra.com/papers/modernmicroprocessors/>
 - <https://www.youtube.com/watch?v=L1ung0wil9Y>

Very Long Instruction Word



Efficient code then and now

- When CPU-s were clocked low, memory access times were roughly up par to ALU command execution times => optimization entailed reducing the number of computations
- Now it has turned around: memory access is orders of magnitudes slower than the CPU. Do as much ALU as you can.
 - And if you are doing too much: LUT up whatever you can :)
- Source:
https://www.theregister.co.uk/2016/04/21/storage_approaches_memory_speed_with_xpoint_and_storageclass_memory/

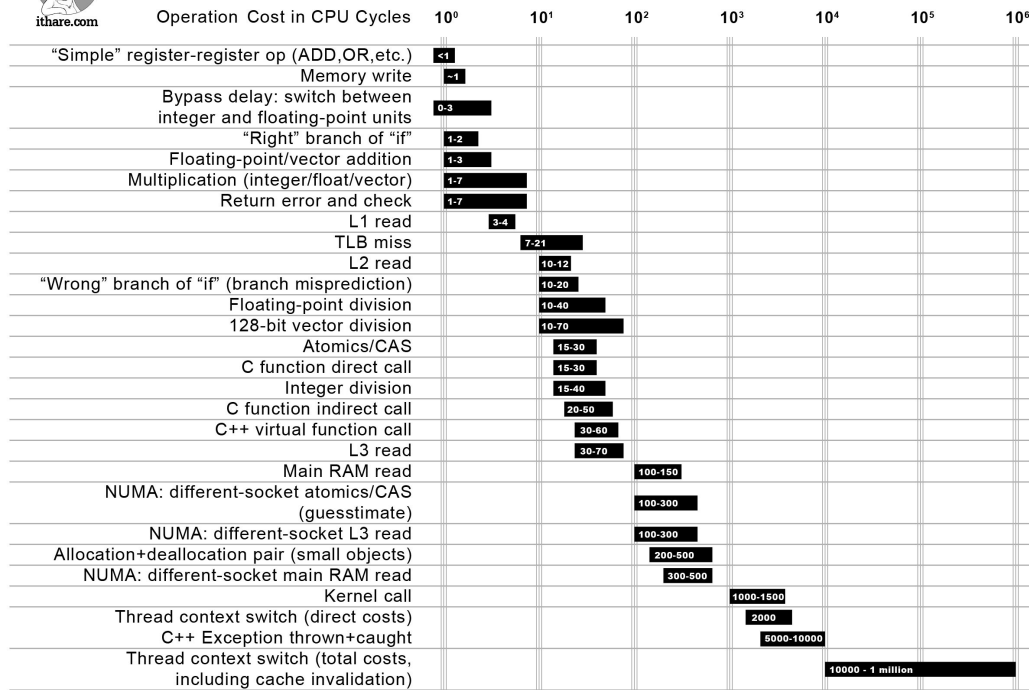
Access times

	Nanoseconds (ns)	Microseconds (μ s)	Milliseconds (ms)	If L1 Access is 1 second
L1 Cache Reference	0.5			1 sec
L2 Cache Reference	7			14 secs
DRAM Access	200			6 mins, 40 secs
Intel Octane 3D XPoint	7,000	7		3 hours, 53 mins, 20 secs
Micron 9100 NVMe PCIe SSD Write	30,000	30		16 hours, 40 mins
Mangstor NX NVMeF Array Write	30,000	30		16 hours, 40 mins
DSSD D5 NVMeF Array	100,000	100		2 days, 7 hours, 33 mins, 20 secs
Mangstor NX NVMeF Array Read	110,000	110		2 days, 13 hours, 6 mins, 40 secs
NVMe PCIe SSD Read	110,000	110		2 days, 13 hours, 6 mins, 40 secs
Micron 9100 NVMe PCIe SSD Read	120,000	120		2 days, 18 hours, 40 mins
Disk Seek	10,000,000	10,000	10	7 months, 10 days, 11 hours, 33 mins, 20 secs
DAS Disk Access	100,000,000	100,000	100	6 years, 4 months, 19 hours, 33 mins, 20 secs
SAN Array Access	200,000,000	200,000	200	9 years, 6 months, 2 days, 17 hours, 20 mins

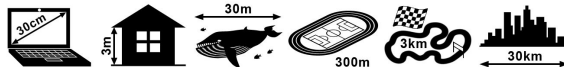
Instruction costs



Not all CPU operations are created equal



Distance which light travels while the operation is performed



Recommended reading

<http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/>



“Integer multiplication/division is quite expensive compared to 'simple' operations above.



“switching between integer and floating-point instructions is not free



“on modern Intel CPUs branch prediction is always dynamic (or at least dominated by dynamic decisions)



“Back in 80s, it was possible to calculate the speed of the program just by looking at assembly.

Sidetrack

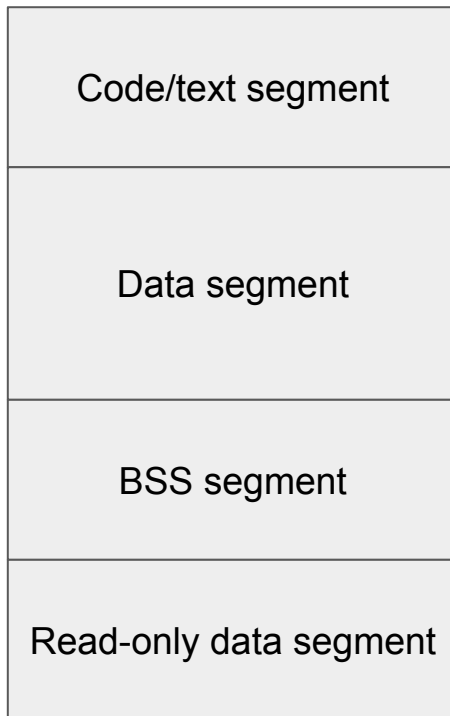
- Why is it so expensive to divide integers? How about floats?
- What algorithms are used to implement these simple operations?
- How about more complex, but frequently used functions (sin, cos, etc.)?
- Check out Jean Michel Muller's books:
 - <https://www.amazon.com/Handbook-Floating-Point-Arithmetic-Jean-Michel-Muller/dp/081764704X>
 - https://www.amazon.com/Elementary-Functions-Implementation-Jean-Michel-Muller/dp/1489979816/ref=sr_1_2?s=books&ie=UTF8&qid=1537995137&sr=1-2

Program layout in memory (C++)

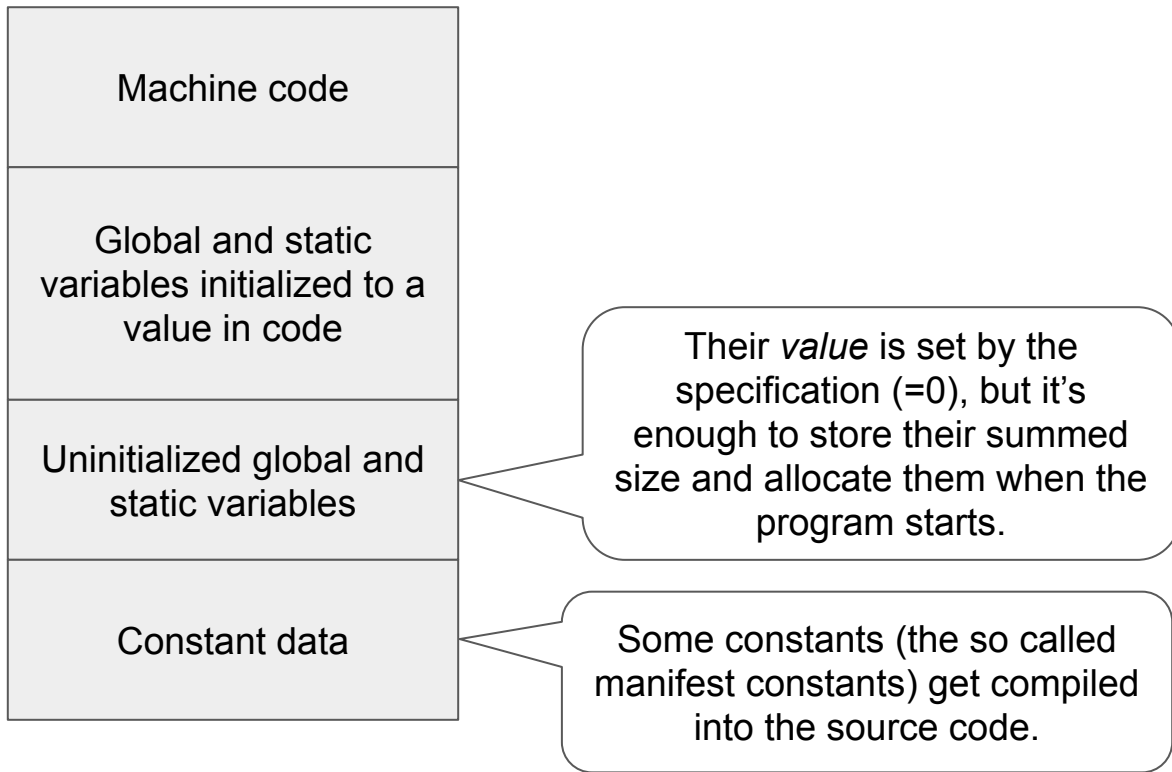
Program in memory

- The executable file can be either an .exe (Windows) or an .elf ([executable and linking format](#) - Unix)
- The compiled and linked source contains the application's **executable image**
- It is a partial image of how our machine code looks in the memory
 - Partial, because e.g. it does not contain the dynamic memory addresses
- The executable image consists of 4 parts

Execution image



Execution image



Endian

- Important to know for multibyte types. Two popular versions:
 - Little endian: least significant bytes have lower memory addresses (=they come first in memory)
 - Big endian: most significant bytes have lower memory addresses
- Mind what you develop on and for what you develop for:
 - Intel CPUs: little endian
 - Wii, Xbox 360, PlayStation 3 (PowerPC variants): big endian

Big-endian		Little-endian	
pBytes + 0x0	0xAB	pBytes + 0x0	0x34
pBytes + 0x1	0xCD	pBytes + 0x1	0x12
pBytes + 0x2	0x12	pBytes + 0x2	0xCD
pBytes + 0x3	0x34	pBytes + 0x3	0xAB

Big- and little-endian representations of the value 0xABCD1234.

Program execution

- Starts with the code at the application's entry point (e.g. `main()`)
- The OS allocates an extra memory for the application's **program stack**
- Every function sub-allocates from this program stack by pushing (and by popping, upon returns) a so called **stack frame**.
 - These allocations are continuous in memory

Stack frame

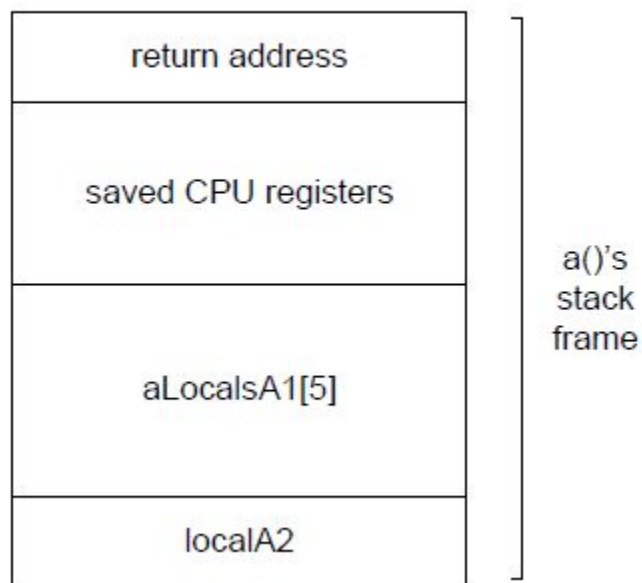
Three types of data are stored in them:

- The memory address of the callee so that the program knows where to resume execution once the function returns
- The contents of the CPU registers when the call was made. Upon returning from the function, these register values are restored.
 - The function return value goes into a special register which we do not overwrite, obviously
- The local variables of the function get allocated here as well

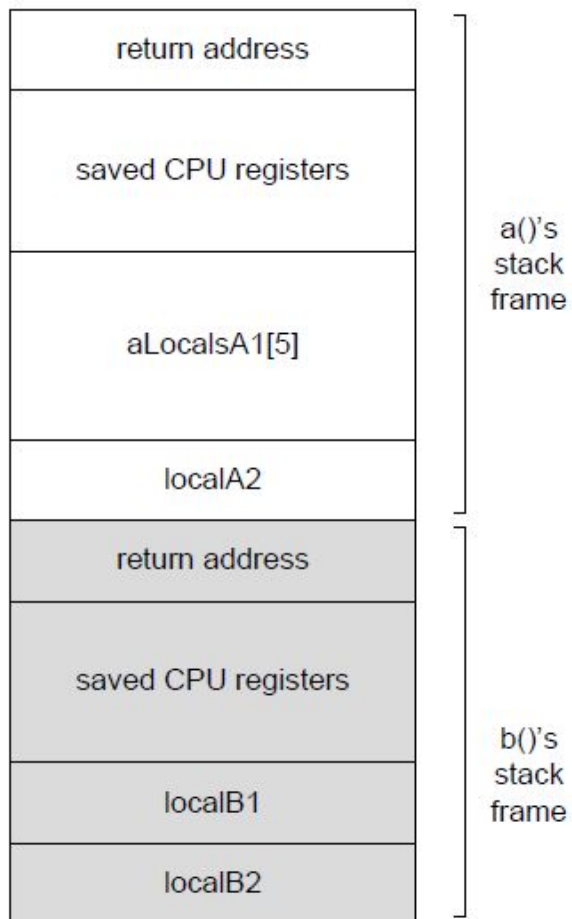
Example

```
void c() { U32 localC1; ... }  
F32 b() {  
    F32 localB1;  
    I32 localB2;...  
    c();  
    return localB1;  
}  
void a() {  
    U32 aLocalsA1[5];  
    ...  
    F32 localA2 = b();  
    ...  
}
```

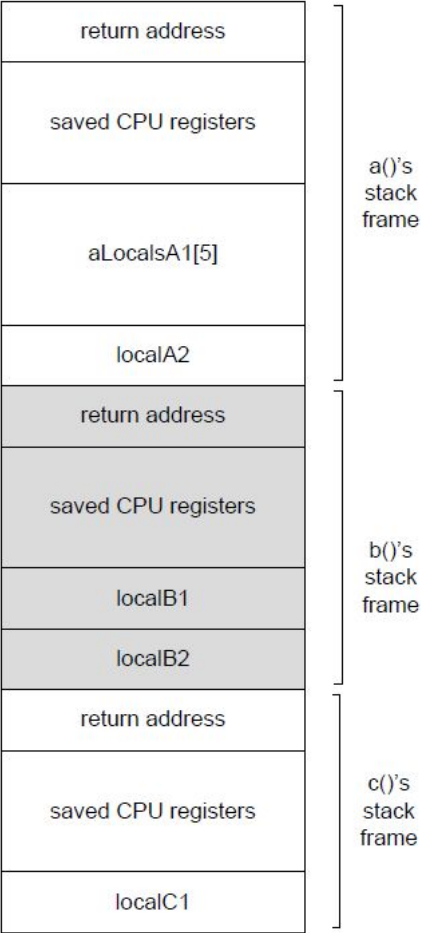
function a() is called



function b() is called



function c() is called



Variables in the memory

- Global and static variables are in the executable image
- Local variables are placed on the stack
- Dynamic variables go to the heap
 - Unfortunately, allocating on the heap is an OS call
 - Which means a potentially long or just unpredictably long wait until the `new*` returns.
 - (*): At least the non in-place new-s.

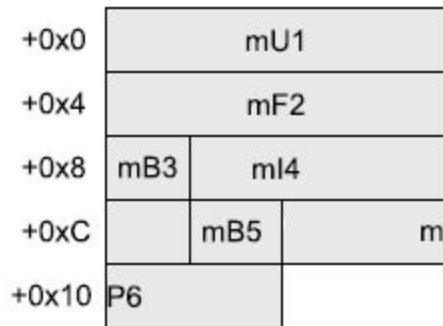
Objects in memory

```
struct Foo  
  
{  
  
    U32 mUnsignedValue;  
  
    F32 mFloatValue;  
  
    I32 mSignedValue;  
  
};
```

+0x0	mUnsignedValue
+0x4	mFloatValue
+0x8	mSignedValue

Objects in memory

```
struct InefficientPacking  
{  
  
    U32 mU1; // 32 bits  
  
    F32 mF2; // 32 bits  
  
    U8 mB3; // 8 bits  
  
    I32 mI4; // 32 bits  
  
    bool mB5; // 8 bits  
  
    char* mP6; // 32 bits  
  
};
```



Objects in memory

```
struct InefficientPacking
{
    U32 mU1; // 32 bits

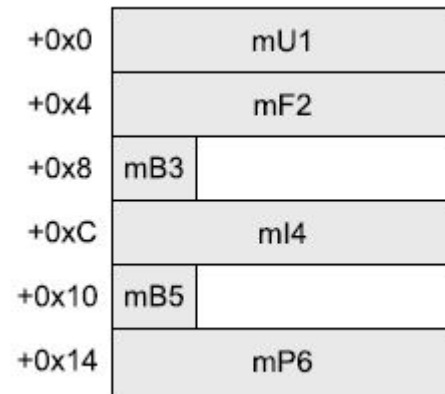
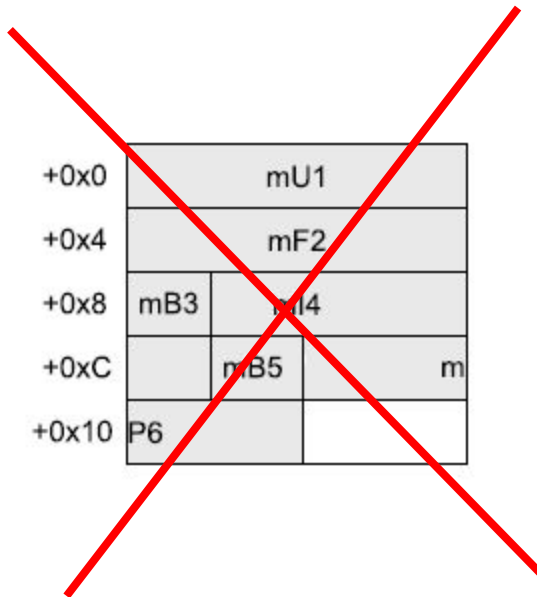
    F32 mF2; // 32 bits

    U8 mB3; // 8 bits

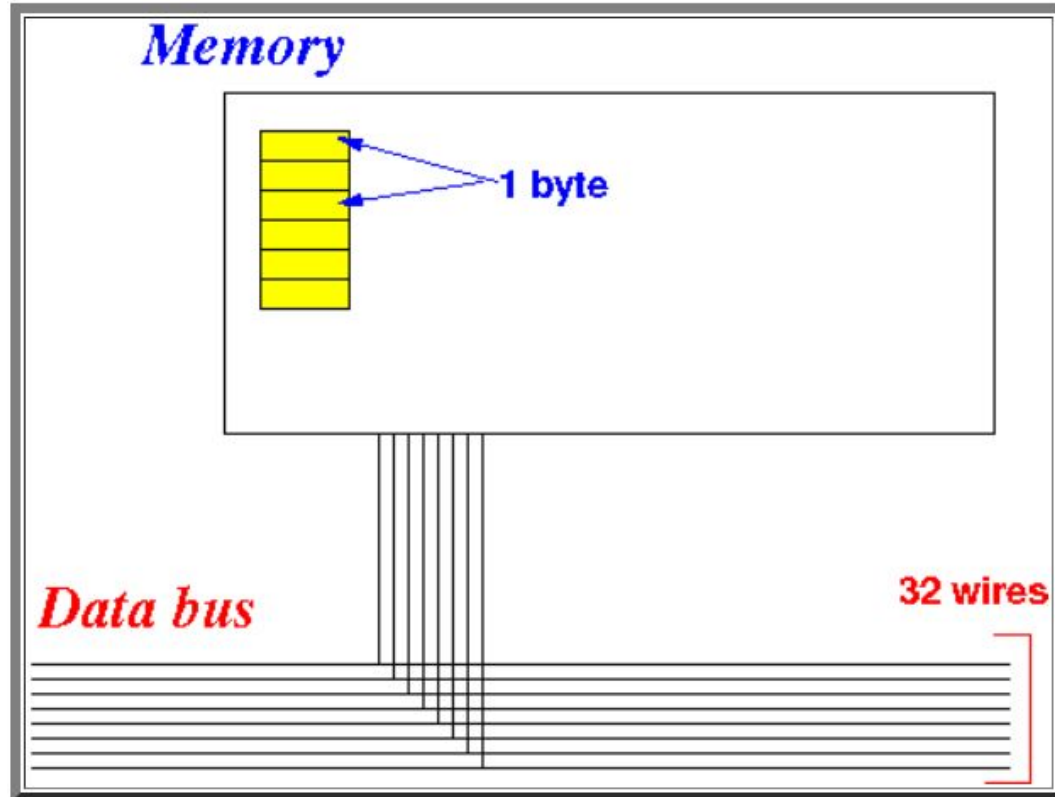
    I32 mI4; // 32 bits

    bool mB5; // 8 bits

    char* mP6; // 32 bits
};
```

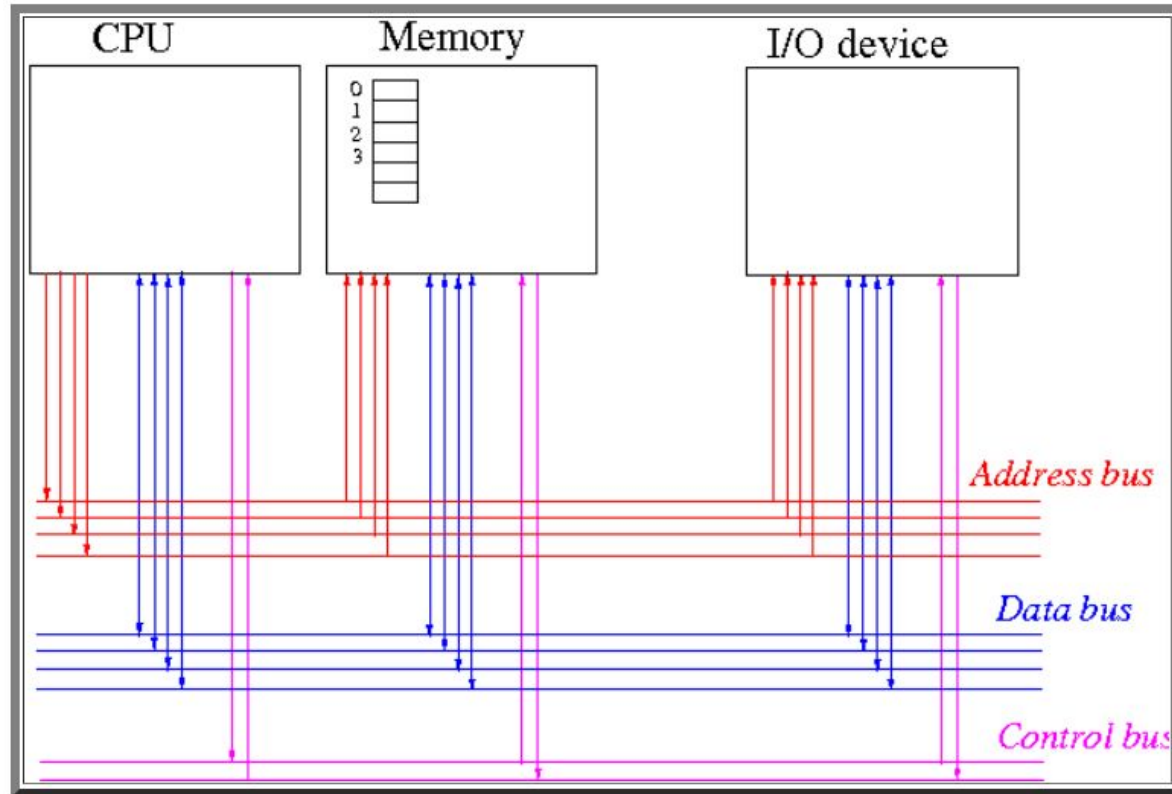


Memory



64 on current generation desktop CPUs.

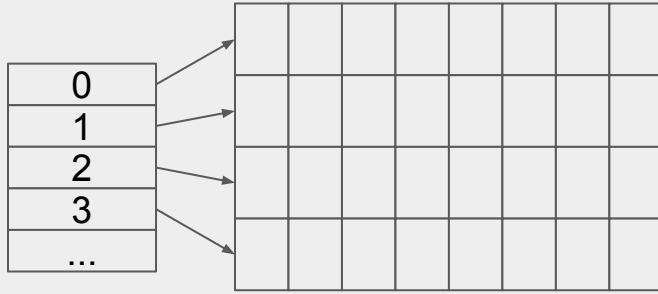
CPU and memory



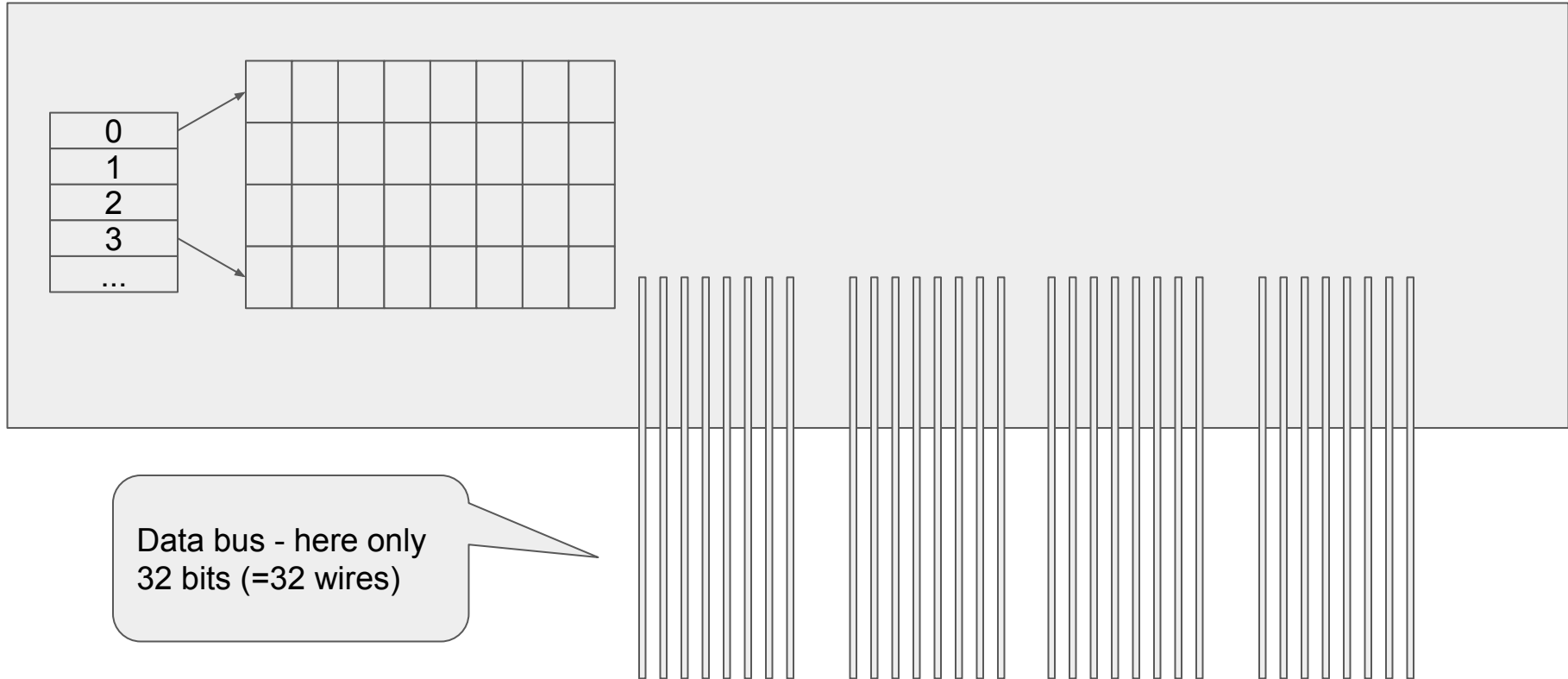
Memory wiring to data bus

0
1
2
3
...

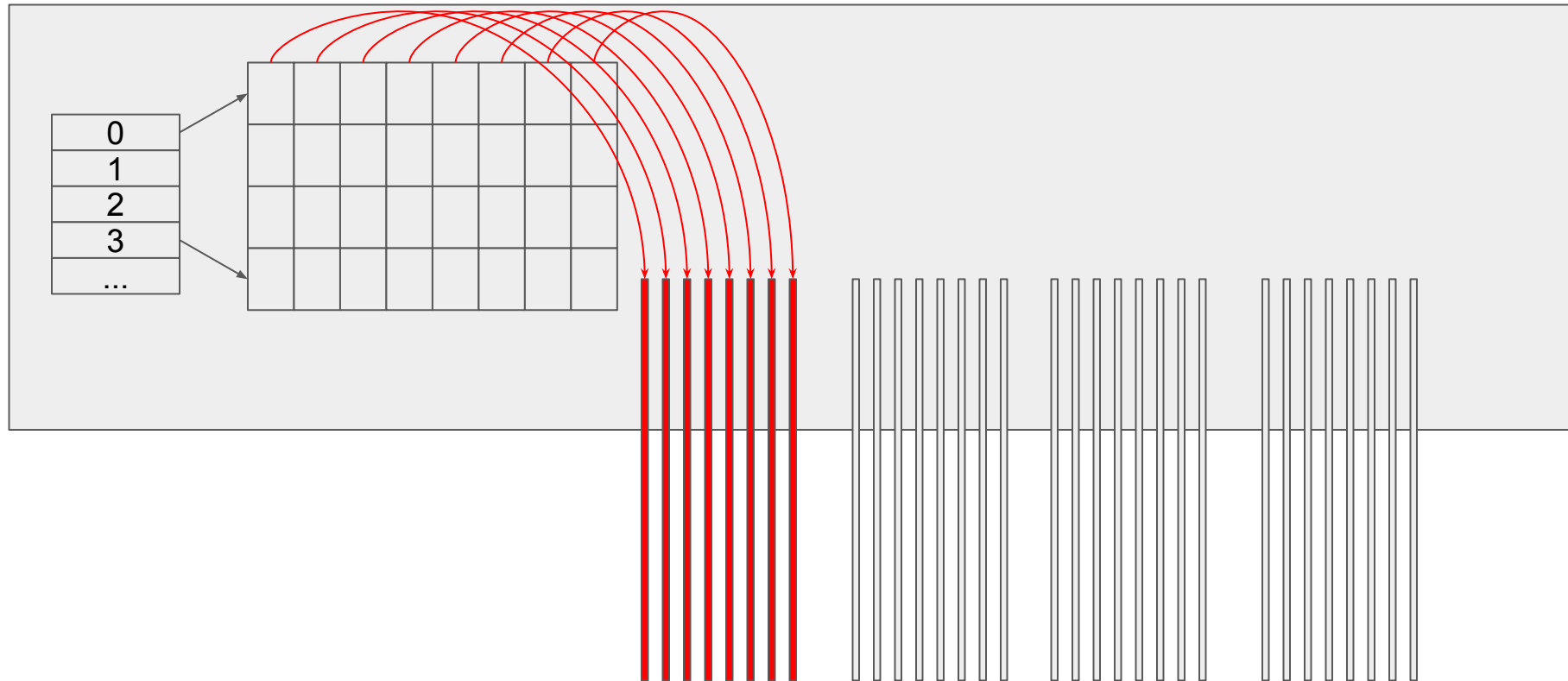
Memory wiring to data bus



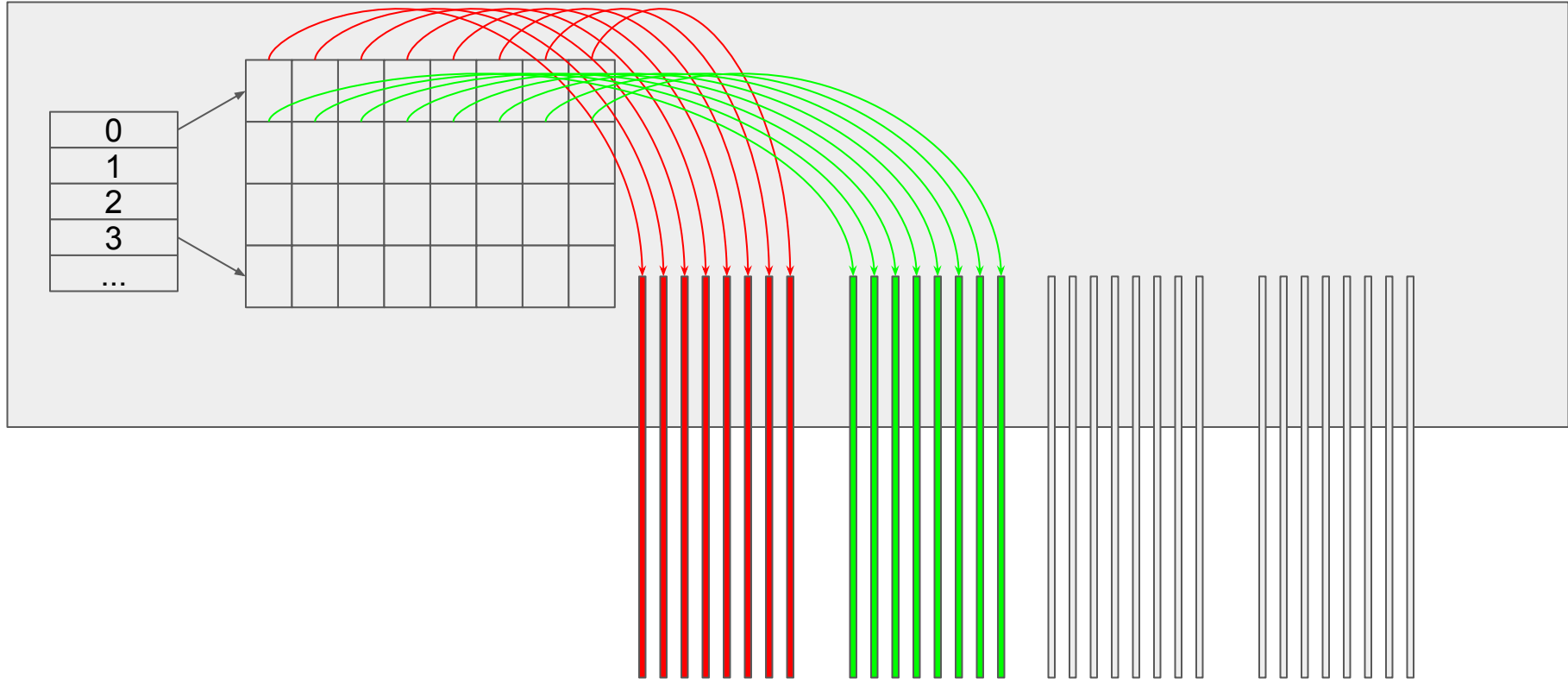
Memory wiring to data bus



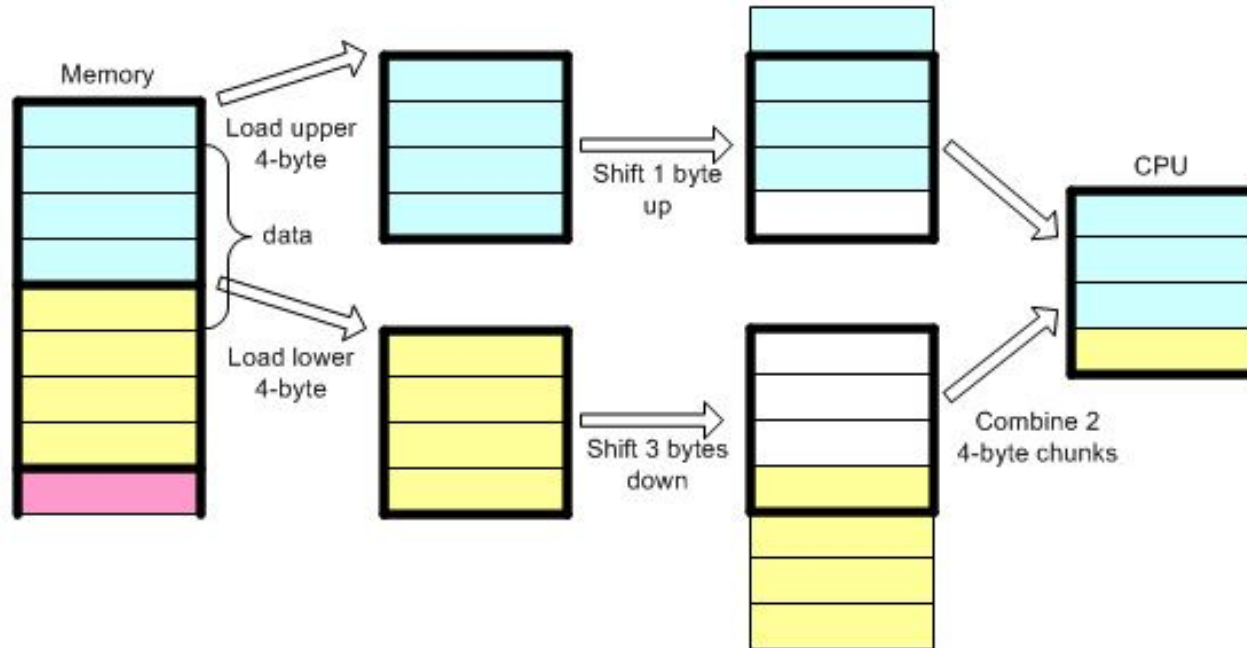
Memory wiring to data bus



Memory wiring to data bus



Reading unaligned data



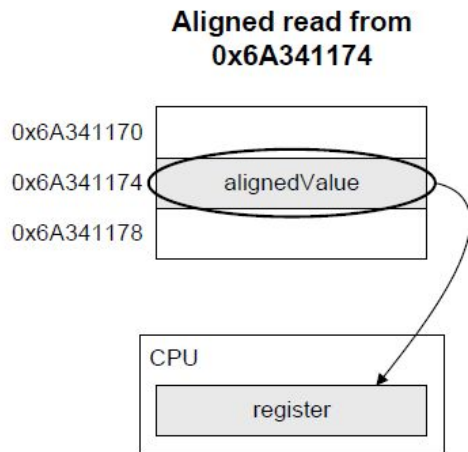
For the interested

Refer to

<http://www.mathcs.emory.edu/~cheung/Courses/255>

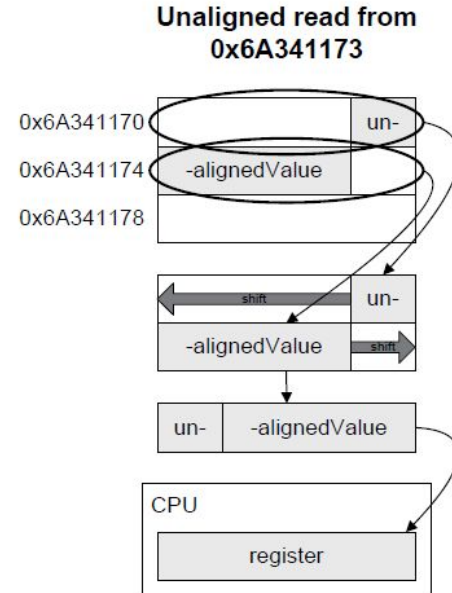
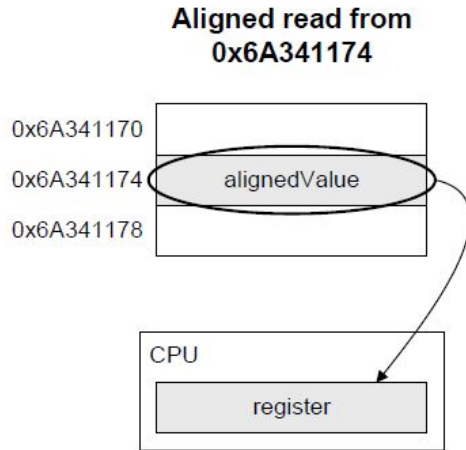
Objects in memory

- Alignment: if the address of the object is a multiple of its size it is **naturally aligned**.



Objects in memory

- Alignment: if the address of the object is a multiple of its size it is **naturally aligned**.



Alignment

- They apply to array/structure elements separately as well!
- +array context padding: the largest alignment of a structure's member variables' determines the alignment of the structure (to improve access when we have an array of the structure)

Caches

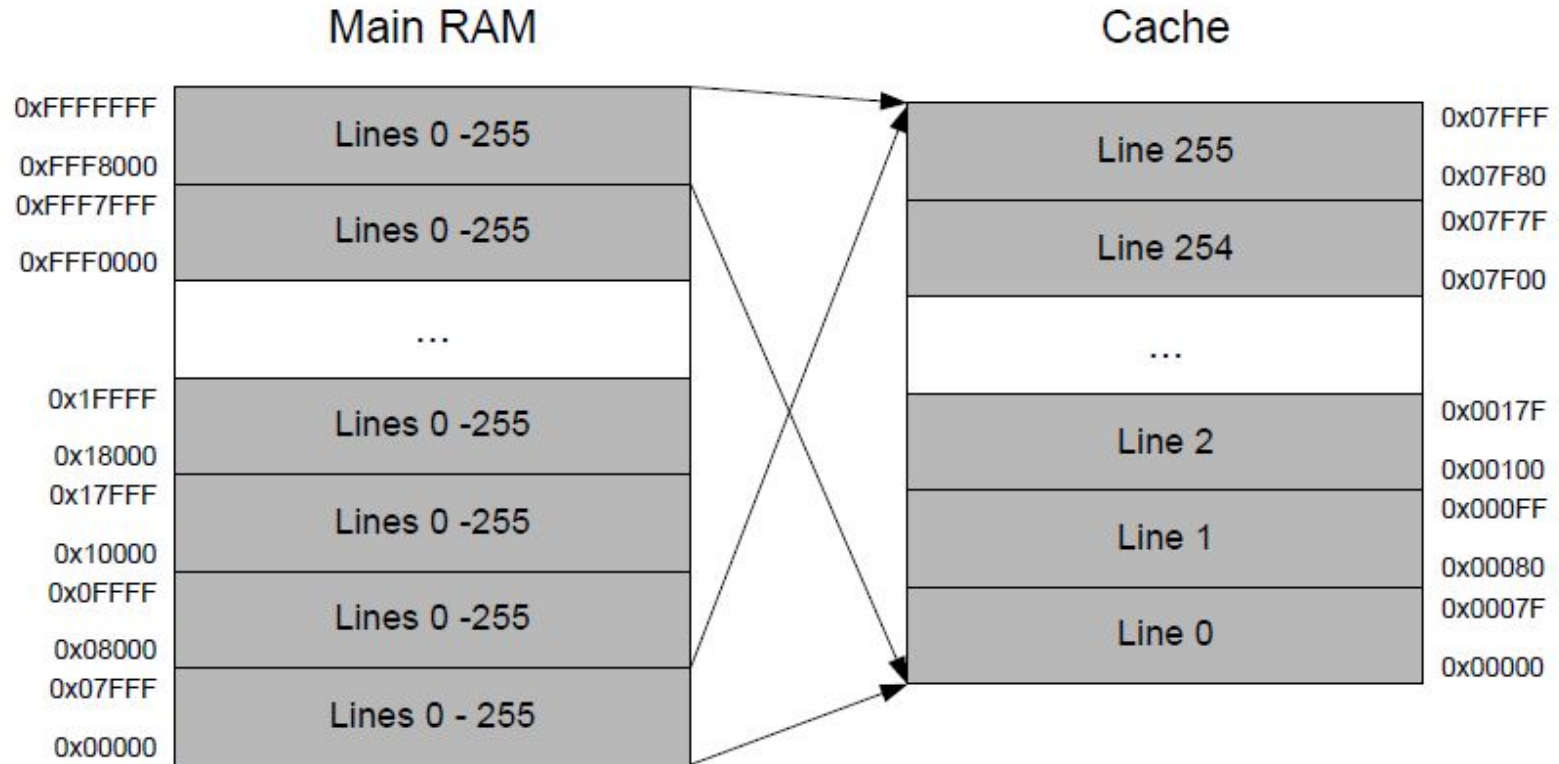
Cache

- A low latency piece of memory for CPU read-write operations
- Two means to achieve that:
 - Cutting edge tech for providing low latency
 - Physically placing this piece of memory close to the CPU
- Simply put, the cache stores a local copy of data from global memory
- If the application reads memory that's in the cache, we get a cache hit (and much lower latency)
- Otherwise, we are facing a cache miss and we have to fetch the data from memory
 - Slow, but we are caching up again

Cache line

- A larger chunk of memory is transferred to the cache when we have a cache miss
 - On i7 architectures the L1, L2, L3 cache lines are 64 bytes
- Sequential reads: way to get cache hits!
- Caches are associative memories: they know what global memory address they contain (via a [translation lookaside buffer](#))

Cache



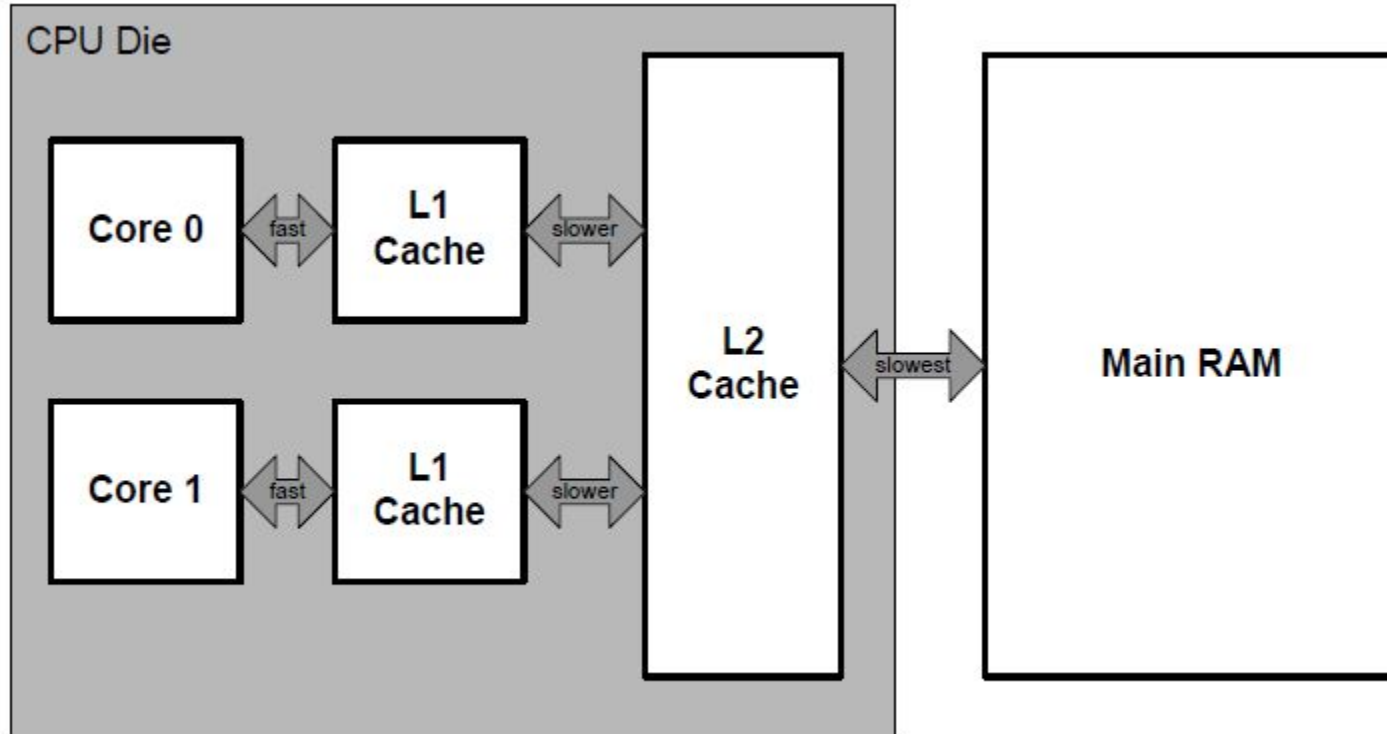
Cache - write policy

- When our program modifies the value of a variable that was in cache, we have to propagate these changes back to the global memory
- The CPU's write policy determines how that happens
- First, write the results into the cache, then
 - Write combine: buffer up changes and send them in a single burst.
 - Write-through: write back the results immediately to the global memory.
 - Write-back: only write back to global memory if certain conditions are met (e.g. cache flush, specific intervals, etc.)*

Hierarchical cache

- Small caches
 - Very fast
 - But many cache misses
- Large caches
 - Not as fast
 - But more cache hits
- So let's have multiple levels of caches with increasing sizes

Caches in multicore systems



Caches in multicore systems

- Cache consistency is an issue: the 'local' caches of the individual cores should store the same value for the same global memory address
- To keep this consistency, there are two common protocols, MESI (modified, exclusive, shared, invalid) and MOESI (modified, owned, exclusive, shared, invalid): https://en.wikipedia.org/wiki/MOESI_protocol

Two main types of caches

- **Instruction cache:** the cache for our machine code itself. Very important to keep this in mind too: branches can mess this up.
- **Data cache:** for caching data when reading from sys mem.
- These two are independent
- There are other caches on GPU-s.

Optimizing for D\$

The data in the memory

- should be laid out sequentially
- as small, as possible
- and we should access them sequentially

Optimizing for I-cache

- Make the machine code for performance-critical loops as small as possible (try to squeeze them into a cacheline)
- Don't call real functions from said loops
 - If we must, see if we can inline that function
 - If we can't inline it (it's too big, etc.), try to define the function such that its instructions are close to our loop in the executable image

Optimizing for I-cache

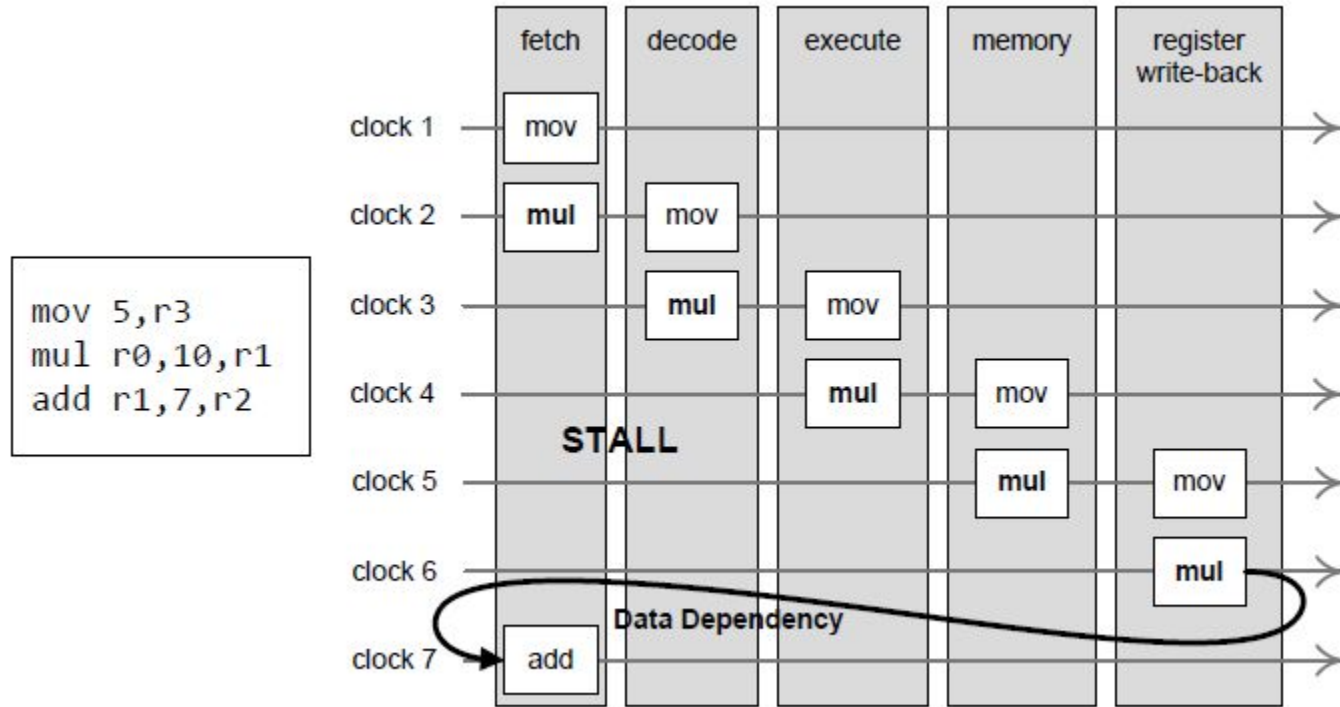
- Easier said than done: the C++ specification does not state where to put the machine code of compiled and linked code!
- Luckily, there are some rules of thumbs that most compilers obey:
 - A function's machine code is almost always sequentially laid out in memory. The exceptions are inlined functions.
 - Functions are put into the executable image in the order of their definition within their translation unit
 - So a single translation unit's functions are laid out in memory sequentially (usually)

Some common issues

Stalling

- When two consecutive instructions depend on each other
- Execution of the latter cannot begin until the former is through the pipeline
- This causes an idle 'bubble' in the pipeline called a data stall

Stalling - data stall



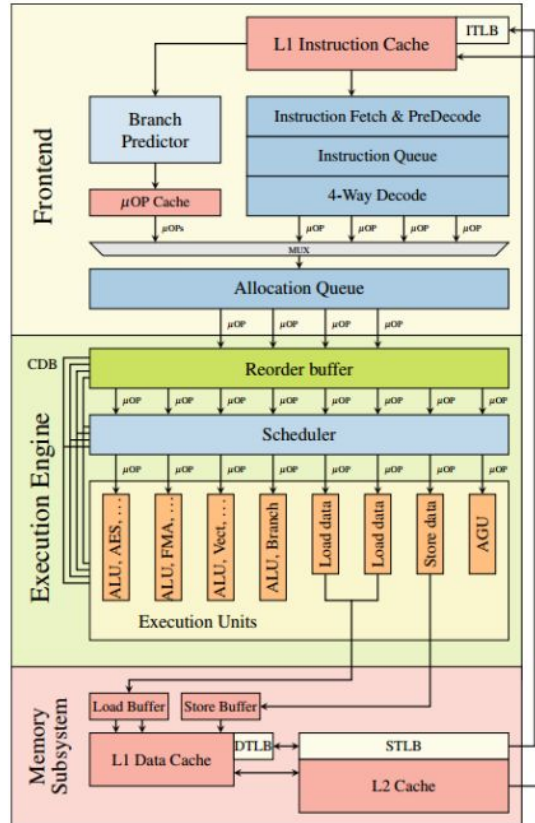
Stalling

- Compiler do their best to shuffle around our code lines to avoid or minimize the effect of such data stalls
- And CPU-s can reorder execution on the fly too

Branch prediction

- When we fetch a branch, the fetch predictor has to guess which branch is taken and fetch data from there
- If it guesses wrong, we have to stall the pipeline until we get the instructions of the correct branch and flush the pipeline (invalidate prior, untaken branch commands)
- Simplest static branching strategy is 'backward branch' (take the branch whose code is on a lower memory address; in case of if-s, the 'then' case, in case of loops, the 'staying in the loop')
 - Rule of thumb: make the 'then' part the road most traveled

Stalling + branching solutions + $O(e) \Rightarrow$ [meltdown](#)



Load-Hit-Store

```
int slow( int * a, int * b) {  
  
    *a = 5;  
  
    *b = 7;  
  
    return *a + *b; // Stall! The compiler doesn't know whether  
                    // a==b, so it has to reload both  
                    // before the add  
  
}
```

Load-Hit-Store

```
int CauseLHS( int *ptrA ) {  
  
    int a,b;  
  
    int * ptrB = ptrA; // B and A point the same direction  
  
    *ptrA      = 5;      // Write data to address ptrA  
  
    b          = *ptrB; // Read that data back again (won't be available for 40/80 cycles)  
  
    a          = b + 10; // Stall! The data b isn't available yet  
  
}
```

You can't case *ptrA because the compile doesn't know if another pointer in the function hasn't modified the value on that address.

Literature

- [Jason Gregory: Game Engine Architecture](#) (many figures are from here)
- Agner: [Optimization manuals](#) (free)
- C++ HPC workshop:
<https://www.youtube.com/watch?v=7xwvLFzRKsk&list=PL1tk5lGm7zvQh6RkurOpDmDOmhB6LzcWL>

Case study: matrix multiplication

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/lecture-slides/MIT6_172F18_lec1.pdf

Matrix multiplication

$$\begin{matrix} \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} & = & \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix} \\ \mathbf{C} & & \mathbf{A} & & \mathbf{B} \end{matrix}$$

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

Naive matrix multiplication

```
for ( int i = 0; i < n; ++i )
{
    for ( int j = 0; j < n; ++j )
    {
        for ( int k = 0; k < n; ++k )
        {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Naive matrix multiplication

```
for ( int i = 0; i < n; ++i )
{
    for ( int j = 0; j < n; ++j )
    {
        for ( int k = 0; k < n; ++k )
        {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Loop order (outer to inner)	Running time (s)
i, j, k	1155.77
i, k, j	177.68
j, i, k	1080.61
j, k, i	3056.63
k, i, j	179.21
k, j, i	3032.82

Naive matrix multiplication

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
    for (int k = 0; k < n; ++k)
      C[i][j] += A[i][k] * B[k][j];
```

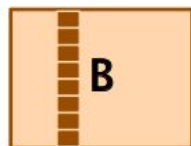
Running time:
1155.77s



=



x



In-memory layout

Excellent spatial locality



Good spatial locality



Poor spatial locality

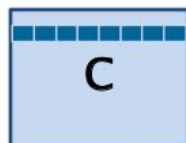


4096 elements apart

Naive matrix multiplication

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

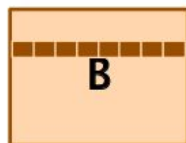
Running time:
177.68s



=



x



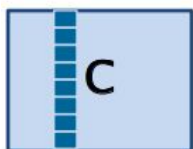
In-memory layout



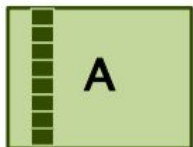
Naive matrix multiplication

```
for (int j = 0; j < n; ++j)
  for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
      C[i][j] += A[i][k] * B[k][j];
```

Running time:
3056.63s



=



x



In-memory layout



Read the full story

- From MIT's Performance Engineering course:
https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/lecture-slides/MIT6_172F18_lec1.pdf
- Spoiler: they really do go from zero (21041.67 seconds) to hero (0.39 seconds)

Case study: evaluating polynomials

<http://lolengine.net/blog/2011/9/17/playing-with-the-cpu-pipeline>

Evaluating polynomials

- Source: <http://lolengine.net/blog/2011/9/17/playing-with-the-cpu-pipeline>
- The problem: how to evaluate a polynomial efficiently?
- Now: Maclaurin approximation to $\sin(x)$:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} - \frac{x^{15}}{15!} + O(x^{16})$$

Evaluating polynomials

```
static double a0 = +1.0;
static double a1 = -1.6666666666666580809419428987894207e-1;
static double a2 = +8.333333333262716094425037738346873e-3;
static double a3 = -1.984126982005911439283646346964929e-4;
static double a4 = +2.755731607338689220657382272783309e-6;
static double a5 = -2.505185130214293595900283001271652e-8;
static double a6 = +1.604729591825977403374012010065495e-10;
static double a7 = -7.364589573262279913270651228486670e-13;
```

```
double sin1(double x)
```

```
{
```

```
    return a0 * x
```

```
        + a1 * x * x * x
```

```
        + a2 * x * x * x * x * x
```

```
        + a3 * x * x * x * x * x * x * x
```

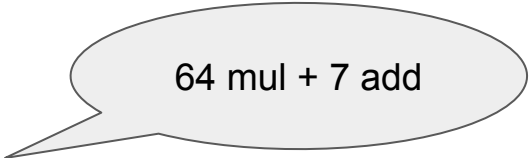
```
        + a4 * x * x * x * x * x * x * x * x * x
```

```
        + a5 * x * x * x * x * x * x * x * x * x * x * x
```

```
        + a6 * x * x * x * x * x * x * x * x * x * x * x * x * x
```

```
        + a7 * x * x * x * x * x * x * x * x * x * x * x * x * x * x * x * x;
```


```
}
```



64 mul + 7 add

Evaluating polynomials

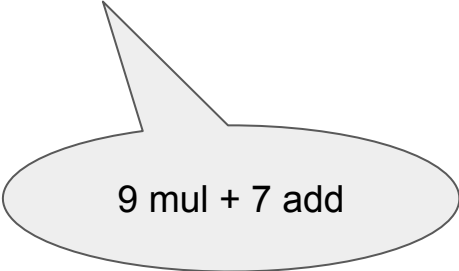
```
double sin2(double x)
{
    double  ret,
           y = x,
           x2 = x * x;
    ret  = a0 * y; y *= x2;
    ret += a1 * y; y *= x2;
    ret += a2 * y; y *= x2;
    ret += a3 * y; y *= x2;
    ret += a4 * y; y *= x2;
    ret += a5 * y; y *= x2;
    ret += a6 * y; y *= x2;
    ret += a7 * y;
    return ret;
}
```



16 mul + 7 add

Evaluating polynomials

```
double sin3(double x) // Horner
{
    double x2 = x * x;
    return x * (a0 + x2 * (a1 + x2 * (a2 + x2 * (a3 + x2 * (a4 + x2 * (a5 + x2 * (a6 + x2 * a7)))))));
}
```



9 mul + 7 add

Benchmarks

Intel® Core™ i7-2620M CPU at 2.70GHz. The functions were compiled using `-O3 -ffast-math`:

function	sin	sin1	sin2	sin3
nanoseconds per call	22.518			

Benchmarks

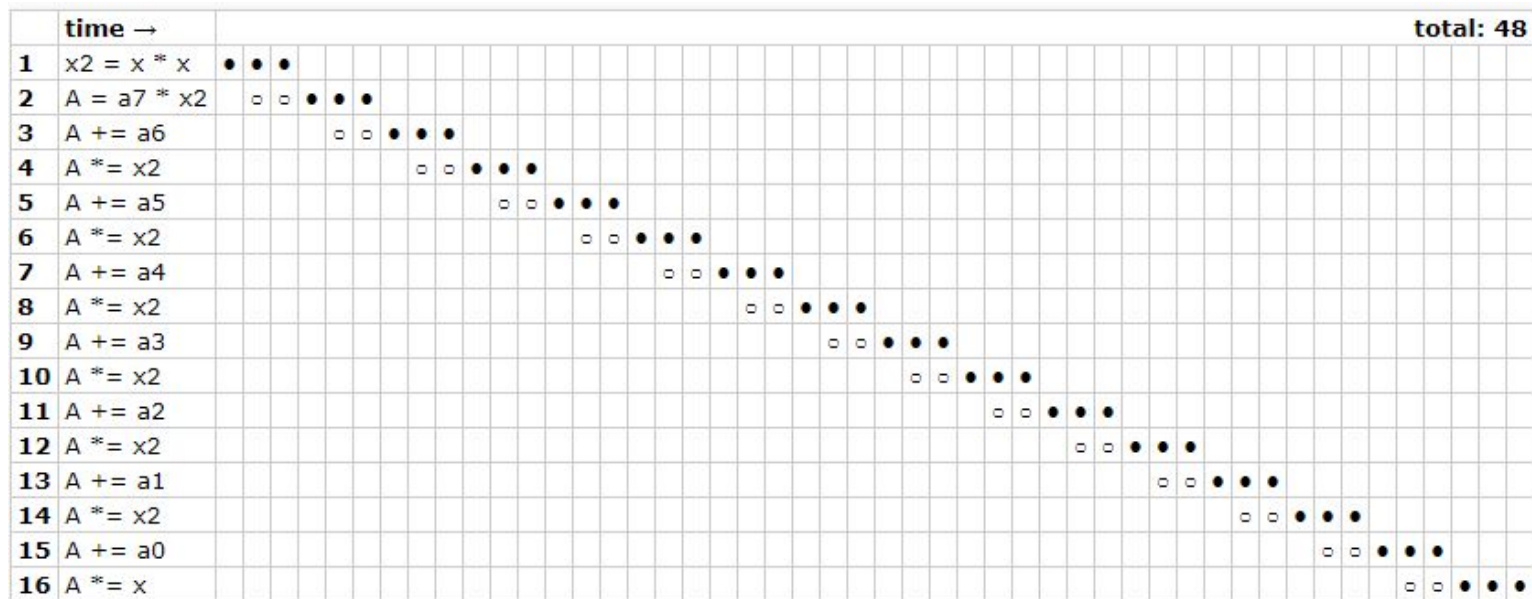
Intel® Core™ i7-2620M CPU at 2.70GHz. The functions were compiled using `-O3 -ffast-math`:

function	<code>sin</code>	<code>sin1</code>	<code>sin2</code>	<code>sin3</code>
nanoseconds per call	22.518	16.406	16.658	25.276



Evaluating polynomials

```
double sin3(double x) // Horner
{
    double x2 = x * x;
    return x * (a0 + x2 * (a1 + x2 * (a2 + x2 * (a3 + x2 * (a4 + x2 * (a5 + x2 * (a6 + x2 * a7))))));
}
```



Evaluating polynomials

```
double sin2(double x)
{
    double  ret,
           y = x,
           x2 = x * x;
    ret  = a0 * y; y *= x2;
    ret += a1 * y; y *= x2;
    ret += a2 * y; y *= x2;
    ret += a3 * y; y *= x2;
    ret += a4 * y; y *= x2;
    ret += a5 * y; y *= x2;
    ret += a6 * y; y *= x2;
    ret += a7 * y;
    return ret;
}
```

	time →																															total: 30		
1	x2 = x * x	●	●	●																														
2	A = a7 * x2	□	□	●	●	●																												
3	x3 = x2 * x				●	●	●																											
4	A += a6					□	●	●	●																									
5	B = a1 * x3							●	●	●																								
6	x5 = x3 * x2								●	●	●																							
7	A *= x2									●	●	●																						
8	C = a2 * x5										□	●	●	●																				
9	B += x											●	●	●																				
10	x7 = x5 * x2												●	●	●																			
11	A += a5													●	●	●																		
12	D = a3 * x7														●	●	●																	
13	B += C															●	●	●																
14	x9 = x7 * x2																●	●	●															
15	B += D																	□	●	●	●													
16	E = a4 * x9																			●	●	●												
17	x11 = x9 * x2																				●	●	●											
18	B += E																					□	●	●	●									
19	A *= x11																							●	●	●								
20	A += B																								□	□	●	●	●					

Evaluating polynomials

- Hand optimizations:

function	sin	sin1	sin2	sin3	sin4	sin5	sin6	sin7
nanoseconds per call	22.518	16.406	16.658	25.276	18.666	18.582	16.366	17.470

- Playing with compiler flags (only -O3):

function	sin	sin1	sin2	sin3	sin4	sin5	sin6	sin7
nanoseconds per call	22.497	30.250	19.865	25.279	18.587	18.958	16.362	15.891

- Or smarten it out: [Estrin scheme](#)
 - AND MEASURE!

Generalization: some common tricks

Bentley rules adapted to modern days

There are various techniques to improve performance:

- Data structures
- Loops
- Logic
- Function

More on this in

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/lecture-slides/MIT6_172F18_lec2.pdf

Closure: context is everything

A famous quote

premature optimization is the root of all evil.

A famous quote: with more context

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3 %. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified. It is

A famous quote: in context

The improvement in speed from Example 2 to Example 2a is only about 12%, and many people would pronounce that insignificant. The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being practiced by pennywise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs. In established engineering disciplines a 12 % improvement, easily obtained, is never considered marginal; and I believe the same viewpoint should prevail in software engineering~ Of course I wouldn't bother making such optimizations on a oneshot job, but when it's a question of preparing quality programs, I don't want to restrict myself to tools that deny me such efficiencies.

There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. **We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.**

Yet we should not pass up our opportunities in that critical 3 %. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail.

http://web.archive.org/web/20130731202547/http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf

A famous quote: in context

- Knuth was actually arguing that manually unrolling the loop body twice so you only have to increase your loop invariant half as many times is an optimization worth taking
- (Mis)quoting Knuth is no excuse for needlessly inefficient code
- Still: the most practical order of things is usually
make it work => make it right => make it fast (if needs be)