Programming with SIMD Instructions

Debrup Chakraborty

Computer Science Department, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional
México D.F., México.
email: debrup@cs.cinvestav.mx

November 13, 2014

Flynn's Taxonomy

A classification of computer architectures by Michael J. Flynn, 1972.

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

SISD

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

- One instruction operating on one data in the same time (traditional sequential processing).
- Flynn includes pipelined architectures also in this category.
- Intel processors $<$ 1996 and AMD $<$ 1998

MISD

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

- Executes different instructions on the same data at the same time.
- This is not common.

SIMD

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

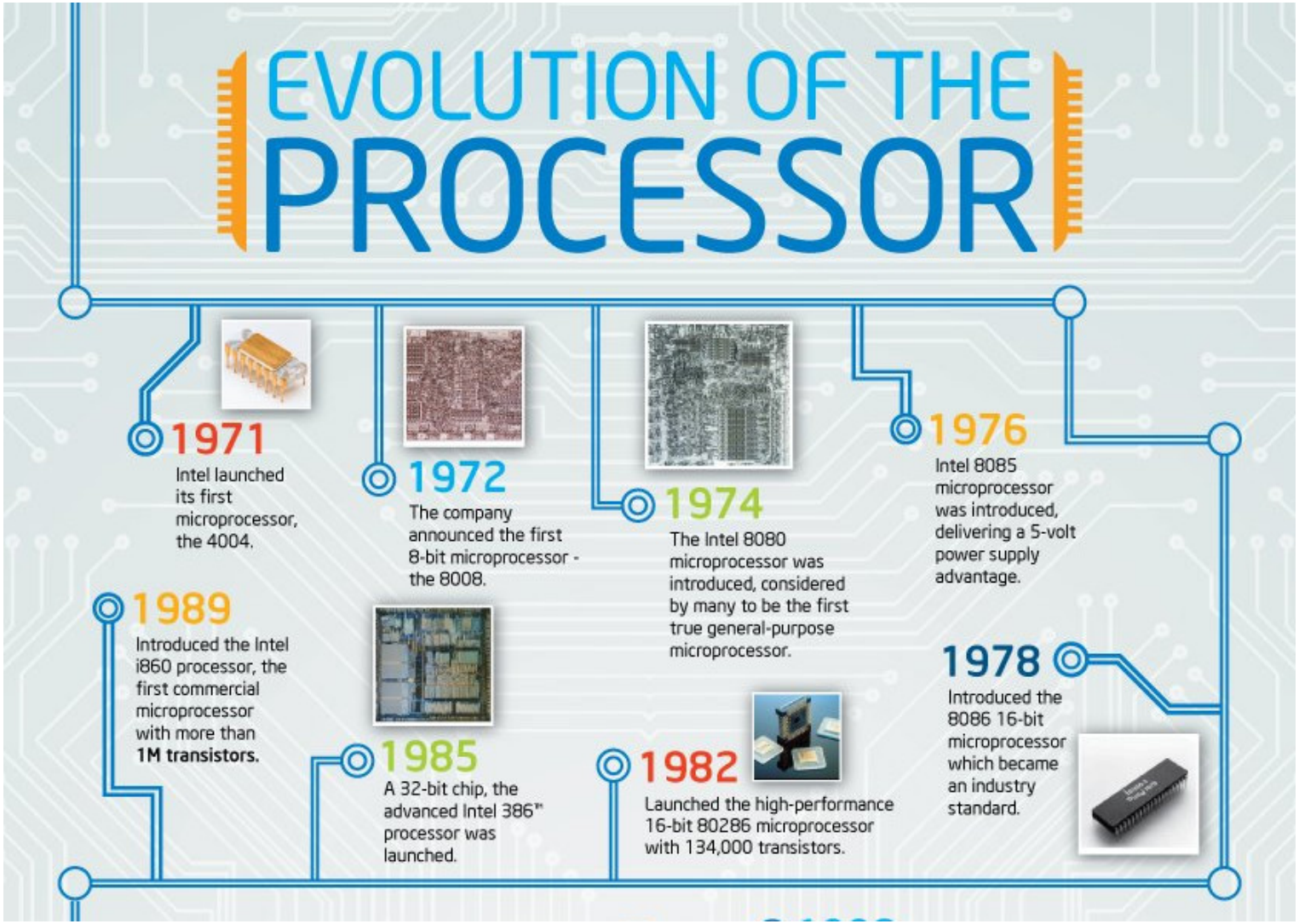
- Execute the same instruction on multiple data at the same time.
- First Intel processor: Intel Pentium MMX (1996), MMX instructions
- First AMD processor : AMD K6-2 (1998), 3DNow! instructions

MIMD

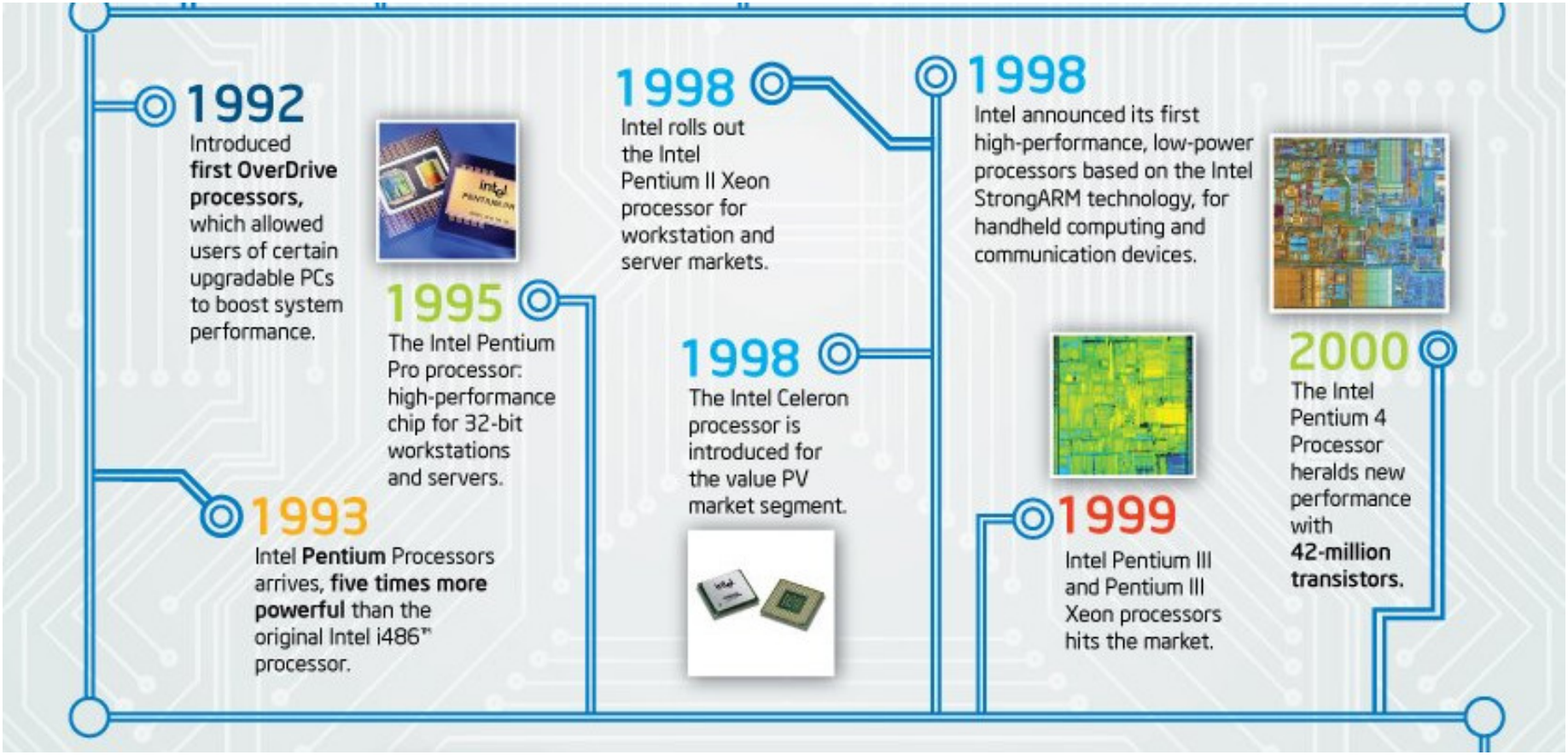
	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

- Executes asynchronously distinct instructions on distinct data.
- Multiprocessor architectures, clusters etc.

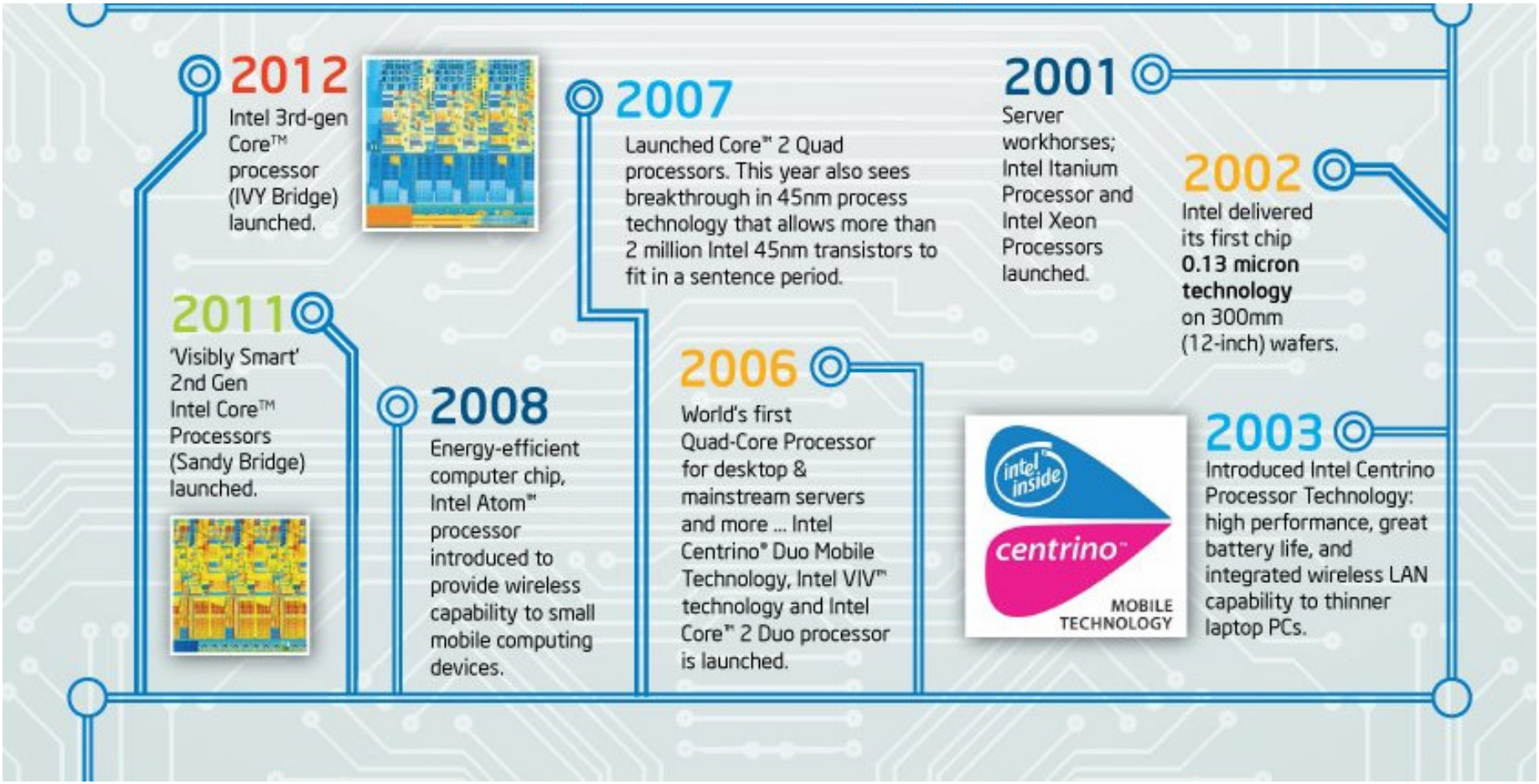
A Brief History of Intel Processors



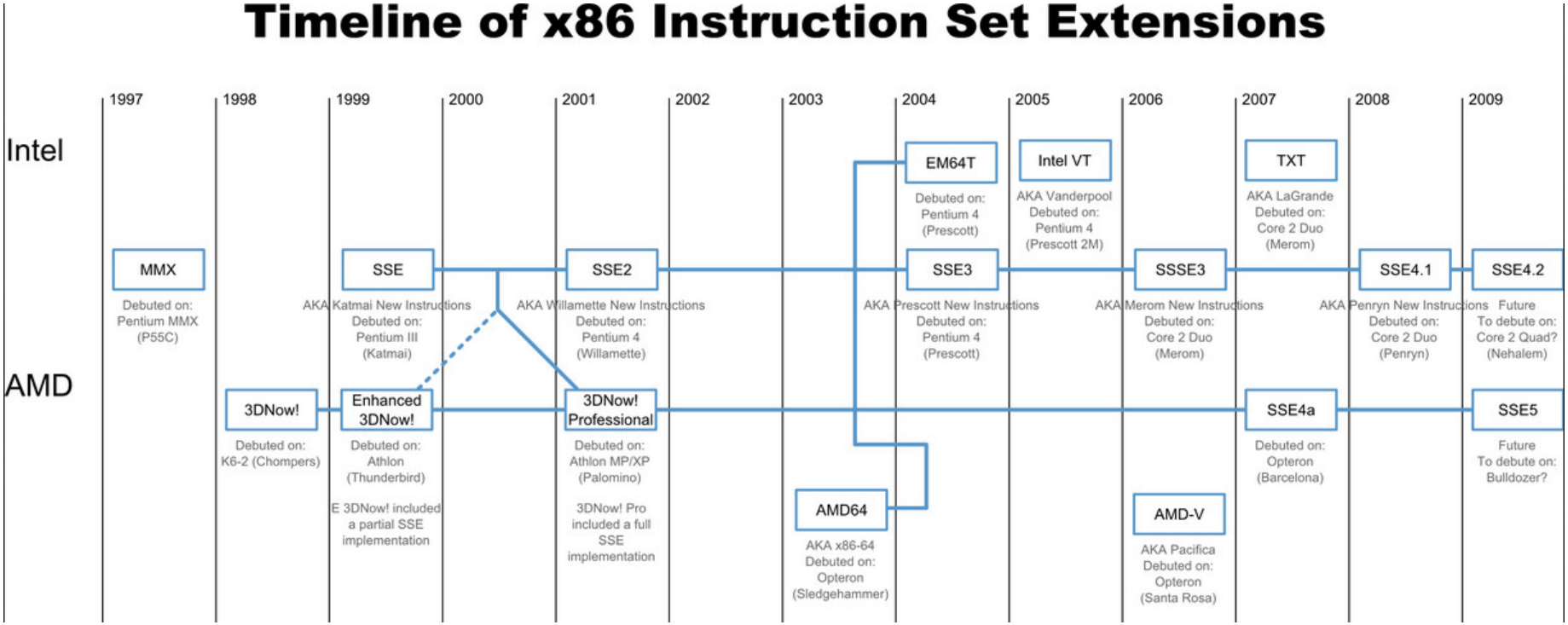
A Brief History of Intel Processors



A Brief History of Intel Processors



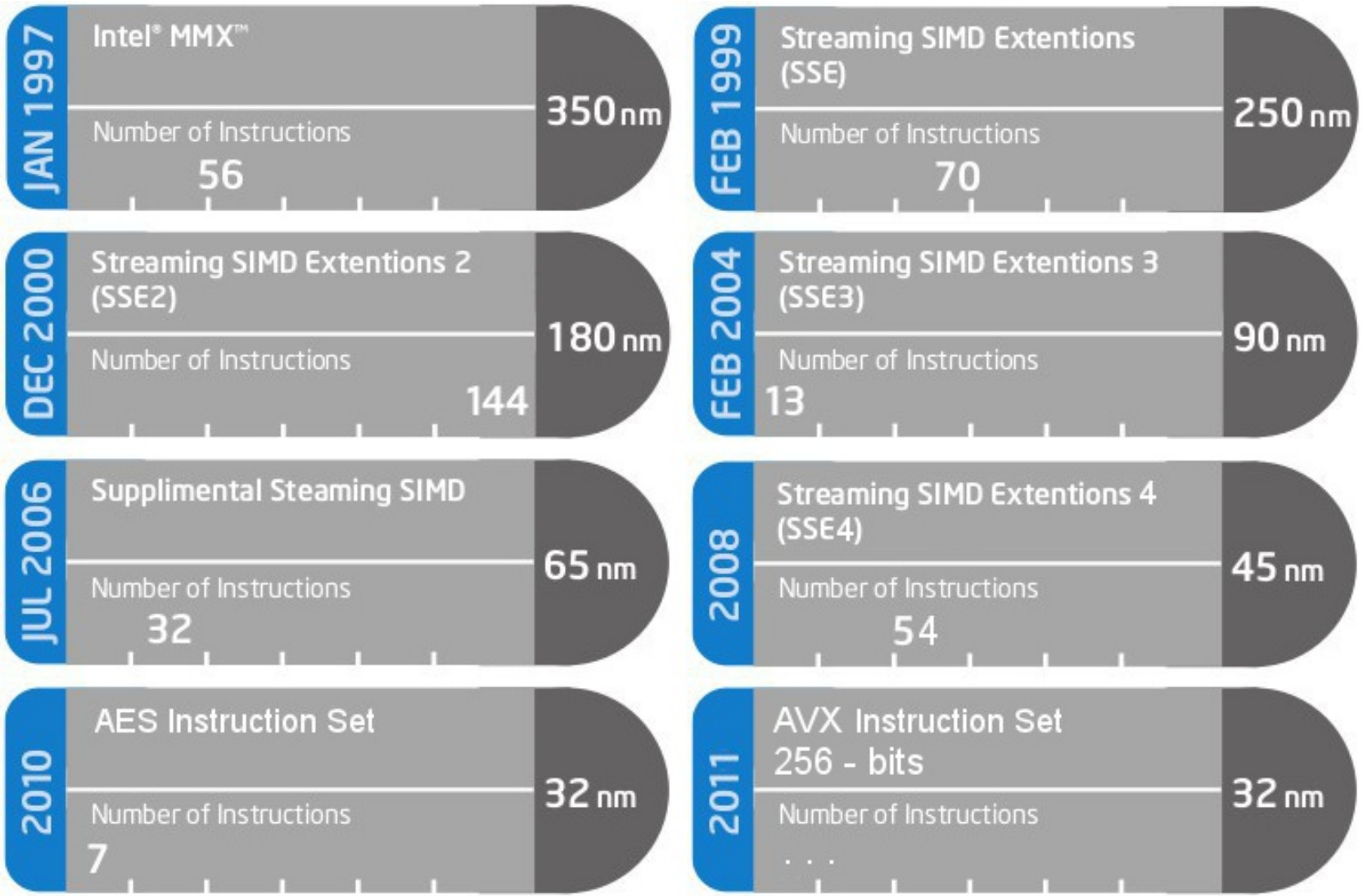
Time line for SIMD Instruction sets



Intel SIMD Instruction Sets

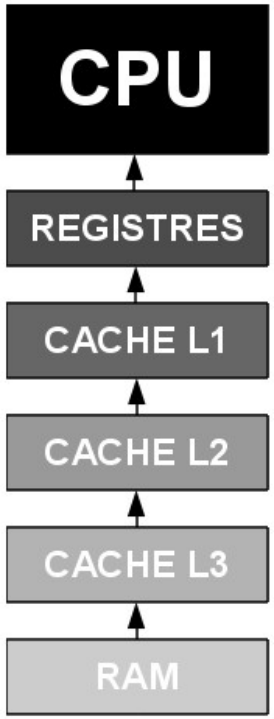
- MMX instructions: Multimedia extentions. 8 registers of 64 bits.
- SSE instructions: Streaming SIMD Extensions. Includes 128 bit registers, and a variety of instructions for bit manipulations, arithmetic etc. Recently includes dedicated instructions for cryptography.
- AVX instructions: Advanced Vectorial Extension, includes 256 bit registers.
- More extensions on the way.

History of SSE



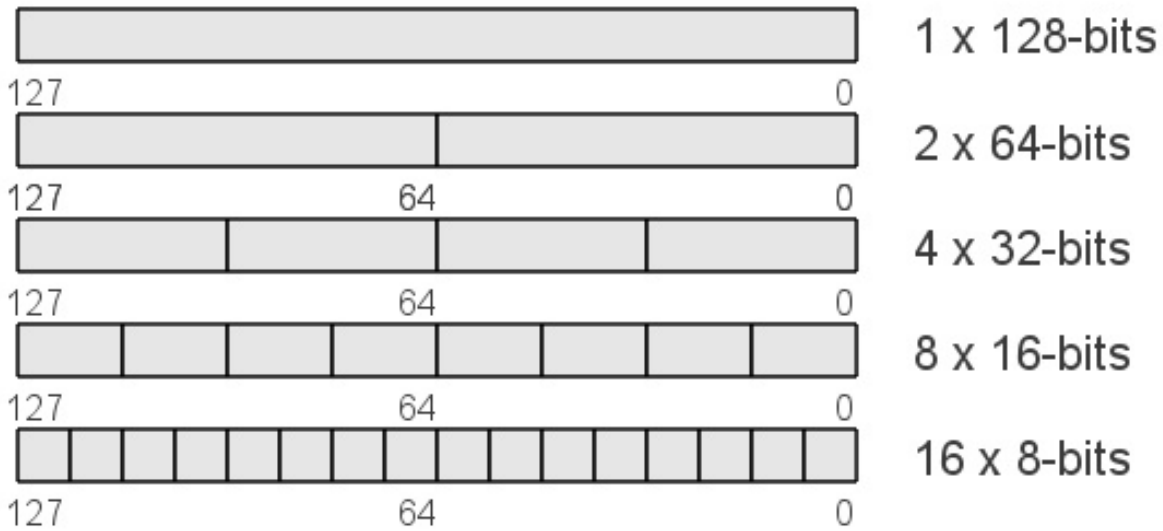
How SSE instructions work?

- Utilize dedicated registers.



How SSE instructions work?

- Multiple data can be packed in a single register



How SSE instructions work?

Task: For each f in array compute

$$f = sqrt(f).$$

SISD:

```
for each f in array {
load f to the floating point register
calculate the square root
write the result from the register to memory
}
```

How SSE instructions work?

SIMD:

```
for each 4 members in array {
load 4 members to the SSE register
calculate 4 square roots in one operation
write the result from the register to memory
}
```

Summary of SSE registers

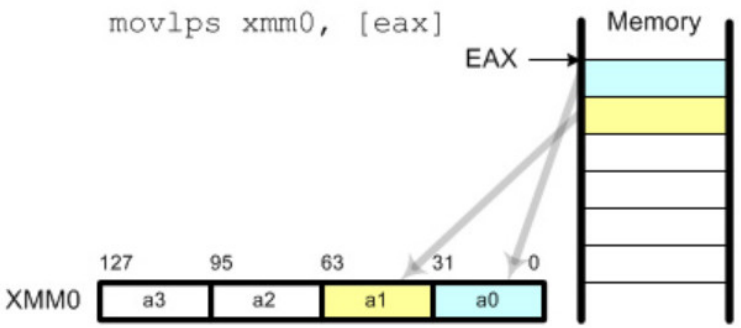
	Number of registers	Size
MMX	8	64-bits
SSE	8	128-bits
SSE2	16	128-bits
...	...	...
AVX	16	256-bits

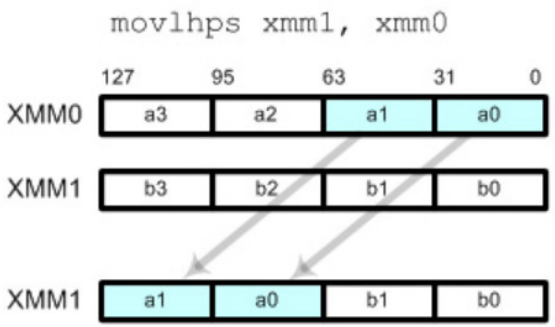
Sample SSE Instructions

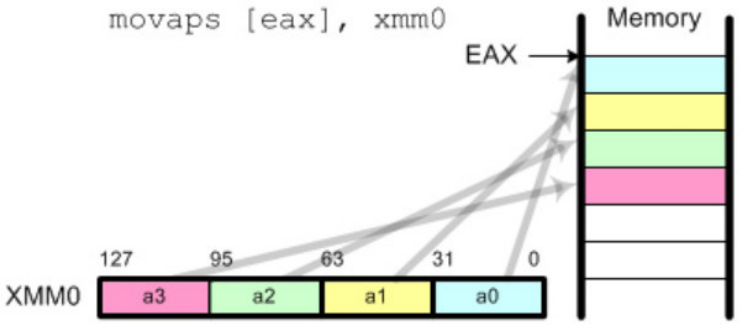
- `movss` xmm, m32
  Load a single-precision (32-bit) floating-point element from memory into the lower of xmm, and zero the upper 3 elements. memory address does not need to be aligned on any particular boundary.
- `movaps` xmm, m128
  Load 128-bits (composed of 4 packed single-precision (32-bit) floating-point elements) from memory into destination. Memory address must be aligned on a 16-byte boundary.
- `movdqa xmm1, m128`,
  Load 128-bits of integer data from memory into destination. Memory address must be aligned on a 16-byte boundary.

(Other usages possible)

Sample SSE Instructions

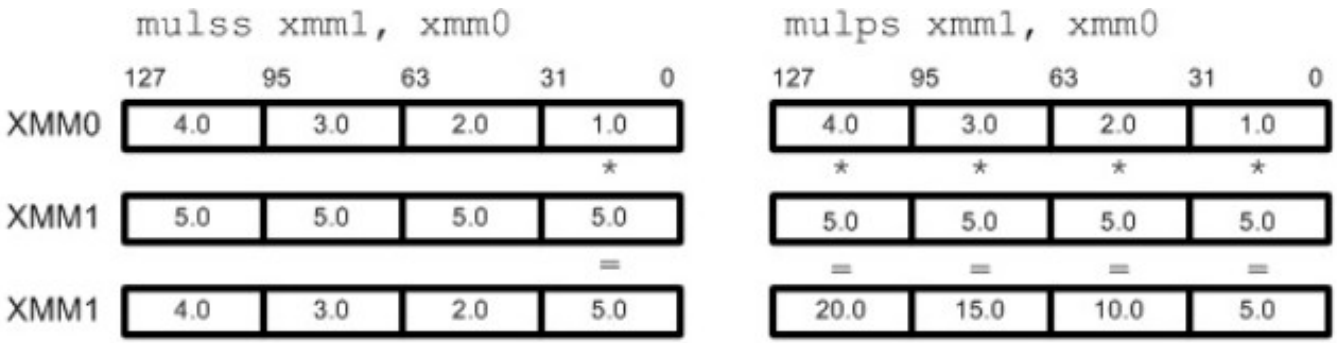






Sample SSE Instructions

- Scalar operations (ss Single scalar)
- Packed (ps Parallel scalar)



Initially it was done by "Inline Assembly":

```
__asm {
        MOV EAX Op_A
        MOV EBX, Op_B

        MOVUPS XMM0, [EAX]
        MOVUPS XMM1, [EBX]

        ADDPS XMM0, XMM1
        MOVUPS [Op_C], XMM0
}
```

Complicated, not very readable, programmer needs to take care of low level details like register allocation etc.

How to use these instructions in my C code?

A better alternative is to use Intel *intrinsics*...

```
__m128 _mm_add_ps(__m128 a , __m128 b );
```

- They are functions coded in assembly in appropriate header files.
- The syntax is much intuitive, and the programmer need not take care of low level details.
- Most compilers (say GCC, ICC) has a good understanding of the intrinsics and can generate optimized codes with them.

What do we need?

- A processor which supports the instructions that we want to use.
- An appropriate copiler, which understand intrinsics(GCC or ICC, in general)
- The headers (.h) which corresponds to the instructions.
- Compile with appropriate flags to enable the instruction sets.
- Know the syntax of the instructions.

Intrinsics

Instructions	Headers	Flags
-MMX	`mmintrin.h`	`-mmmx`
-SSE	`xmmintrin.h`	`-msse`
-SSE2	`emmintrin.h`	`-msse2`
-SSE3	`pmmintrin.h`	`-msse3`
-SSSE3	`tmmintrin.h`	`-mssse3`
-SSE4.1 et SSE4.2	`smmintrin.h`	`-msse4.1 -msse4.2`
-AES et PCLMUL	`wmmintrin.h`	`-maes -mpclmul`

Intrinsics

- In intrinsics we can use some nonstandard data types : `__m128d`, `__m128i`
- General syntax for function names: `_mm_<name>_<type>`
  - The prefix `_mm_` is always present
  - The second part is `<name>`, generally it is same as the assembly mnemonic, but not always.
  - The final part `<type>` indicates the packing information.

Intrinsics

Examples :

```
__m128i _mm_add_epi8(__m128i a, __m128i b)
__m128i _mm_add_epi32(__m128i a, __m128i b)
__m128i _mm_add_epi64(__m128i a, __m128i b)

__m128i _mm_and_si128(__m128i a, __m128i b)
__m128i _mm_xor_si128(__m128i a, __m128i b)
__m128i _mm_or_si128(__m128i a, __m128i b)
```

`_piX` : vector MM (64-bits) packed with X-bit words
`_epiX` : vector XMM (128-bits) packed with X-bit words
`_si64` : vector MM (64-bits) of a single 64-bit word
`_si128` : vector XMM (128-bits) of a single 128-bit word
see instruction list at:
`https://software.intel.com/sites/landingpage/IntrinsicsGuide/`