Single Instruction Multiple Data (SIMD)

- Motivation
  - SIMD Multimedia Extensions started with the simple observation that many media applications operate on narrower data types than the 32-bit processors were optimized for
  - Many graphics systems used 8 bits to represent each of the three primary colors plus 8 bits for transparency
  - Audio samples are usually represented with 8 or 16 bits
- The additional cost of such partitioned adders was small

Instruction category	Operands
Unsigned add/subtract	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Maximum/minimum	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Average	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Shift right/left	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Floating point	Sixteen 16-bit, eight 32-bit, four 64-bit, or two 128-bit

Example SIMD extensions for 256-bit-wide operations

SIMD vs. Vector Architectures

Fixed vs. Variable Length

SIMD extensions fix the number of data operands in the opcode, leading to hundreds of instructions in x86 extensions. Vector architectures use a vector length register to specify operand count.

Limited Addressing Modes

SIMD lacks sophisticated addressing modes of vector architectures (strided accesses and gather-scatter), limiting compiler vectorization capabilities.

Conditional Execution

SIMD usually doesn't offer mask registers to support conditional execution of elements, unlike vector architectures.

These limitations make it harder for compilers to generate SIMD code and increase the difficulty of programming in SIMD assembly language.

Advantages

- Despite of their weaknesses, the SIMD extensions is gaining popularity because
  - They cost little to add to the standard arithmetic unit and they were easy to implement
  - They require little extra state compared to vector architectures, which is always a concern for context switch times
  - You need a lot of memory bandwidth to support a vector architecture, which many computers don't have
  - SIMD does not have to deal with problems in virtual memory when a single instruction that can generate 64 memory accesses can get a page fault in the middle of the vector. SIMD extensions use separate data transfers per SIMD group of operands that are aligned in memory, and so they cannot cross page boundaries.

Evolution of x86 SIMD Extensions

1

MMX (1996)

Repurposed 64-bit floating-point registers for 8×8-bit or 4×16-bit operations. Added parallel MAX/MIN operations and DSP-like instructions.

2

SSE (1999)

Added separate 128-bit registers for 16×8-bit, 8×16-bit, or 4×32-bit operations. Added single-precision floating-point arithmetic.

3

SSE2-SSE4 (2001-2007)

Added double-precision SIMD floating-point and specialized multimedia instructions.

4

AVX (2010)

Doubled register width to 256 bits, doubling operations on all data types. Prepared for future 512-bit and 1024-bit extensions.

Example

```
        L.D     F0,a          ;load scalar a
        MOV     F1, F0        ;copy a into F1 for SIMD MUL
        MOV     F2, F0        ;copy a into F2 for SIMD MUL
        MOV     F3, F0        ;copy a into F3 for SIMD MUL
        DADDIU  R4,Rx,#512    ;last address to load
Loop:   L.4D    F4,0(Rx)      ;load X[i], X[i+1], X[i+2], X[i+3] to F4, F5, F6 and F7
        MUL.4D  F4,F4,F0      ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4D    F8,0(Ry)      ;load Y[i], Y[i+1], Y[i+2], Y[i+3] to F8, F9, F10 and F11
        ADD.4D  F8,F8,F4      ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
        S.4D    F8,0(Rx)      ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU  Rx,Rx,#32     ;increment index to X
        DADDIU  Ry,Ry,#32     ;increment index to Y
        DSUBU   R20,R4,Rx     ;compute bound
        BNEZ    R20,Loop      ;check if done
```