CSC 411

Computer Organization (Spring 2024)
Lecture 17: Arithmetic operations and SIMD instructions

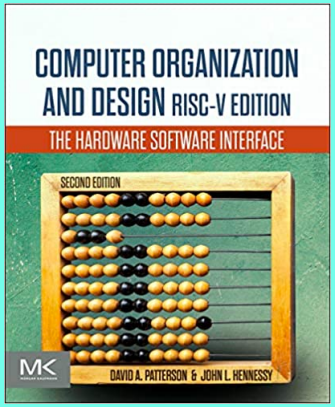
Prof. Marco Alvarez, University of Rhode Island

Disclaimer

Some figures and slides are adapted from:

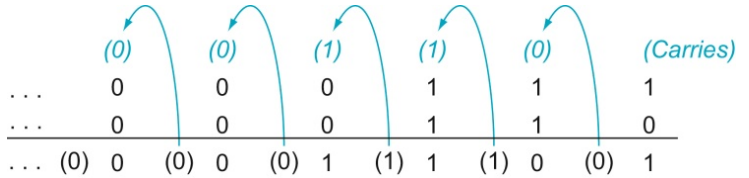
Computer Organization and Design (Patterson and Hennessy)

The Hardware/Software Interface



Addition/subtraction

Integer addition



- Overflow
  - two positive operands, overflow if result's most significant bit is 1
  - two negative operands, overflow if result's most significant bit is 0
  - positive and negative operands, no overflow
- Subtraction
  - just add negative/positive counterpart of the second operand — invert all bits and add 1

Practice

- Using 8 bits add:

0 0 1 1 1 0 0 1
0 1 1 0 0 1 0 0

- Did overflow happened?

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Multiplication

Warming-up

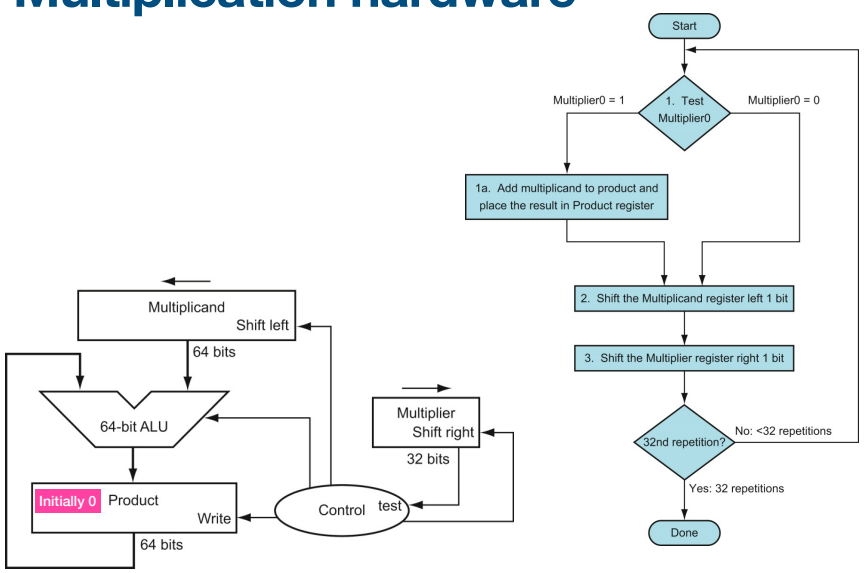
Multiplicand					1	0	0	0
Multiplier				X	1	0	0	1
					1	0	0	0
				0	0	0	0	
			0	0	0	0		
		1	0	0	0			
Product	1	0	0	1	0	0	0	

Practice

- Using 4 bits multiply:

1 0 1 1
0 0 1 0

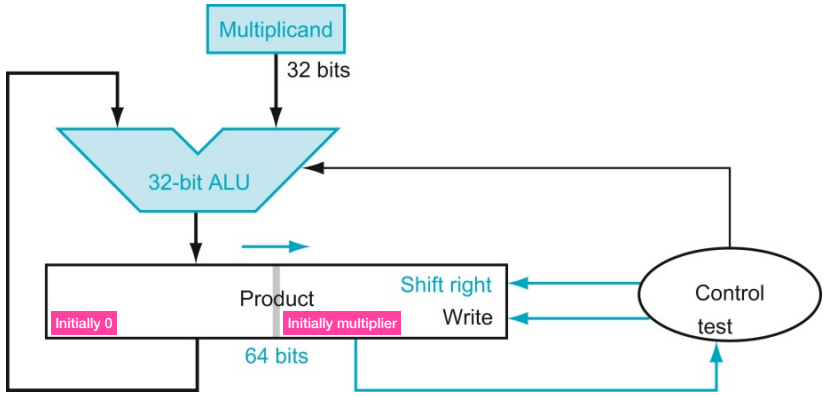
Multiplication hardware



Example

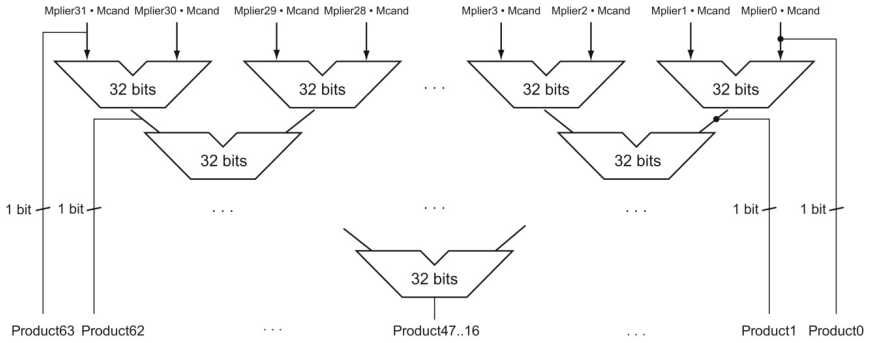
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Better multiplication



One cycle per partial-product addition

Fast multiplication



Rather than use a single 32-bit adder 31 times, this hardware "unrolls the loop" to use 31 adders and then organizes them to minimize delay. Can be pipelined.

Division

Warming-up

Quotient					1	0	0	1			
Divisor/Dividend	1	0	0	0	1	0	0	1	0	1	0
					1	0	0	0			
								1	0		
								1	0	1	
								1	0	1	0
							-	1	0	0	0
Remainder										1	0

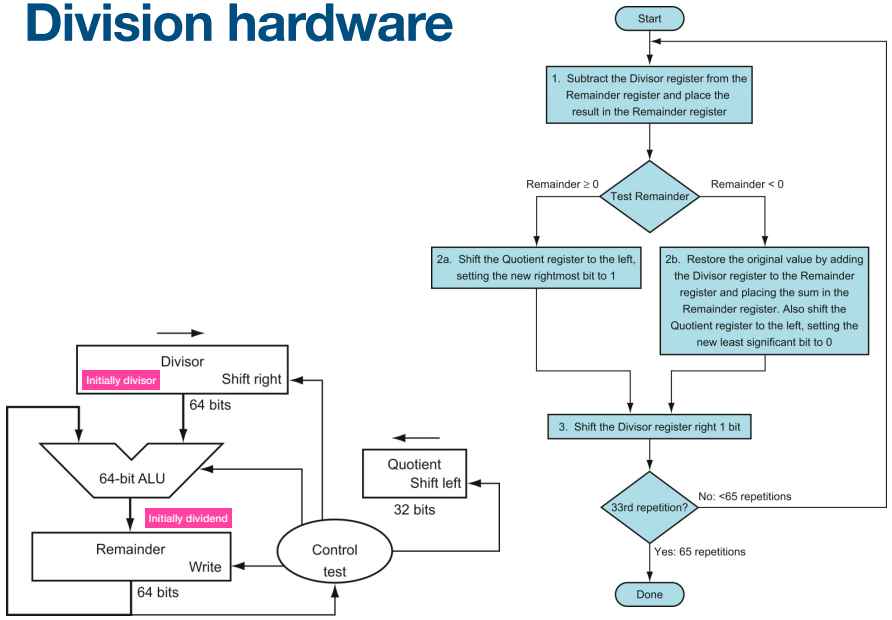
Practice

- Divide 1 0 1 1 0 0 by 1 1

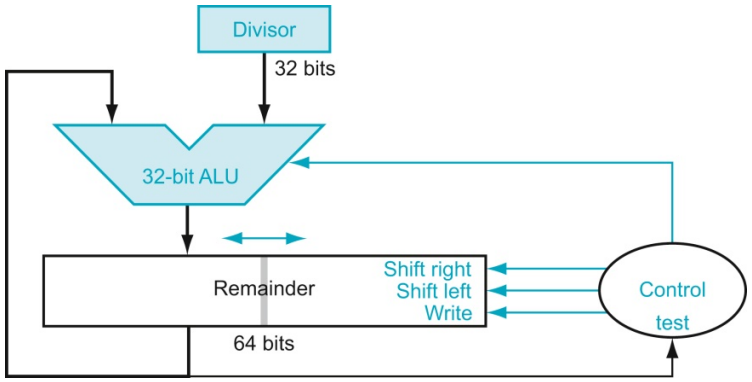
Division

- Check for 0 divisor
- Long division approach
  - if divisor <= dividend bits
    - 1 bit in quotient, subtract
  - otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - divide using absolute values
  - adjust sign of quotient and remainder as required

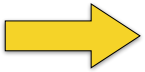
Division hardware



Improved hardware



Very similar to a multiplier, in fact, same hardware can be used for both. Can't use parallel hardware as in multiplier (subtraction is conditional on sign of remainder)



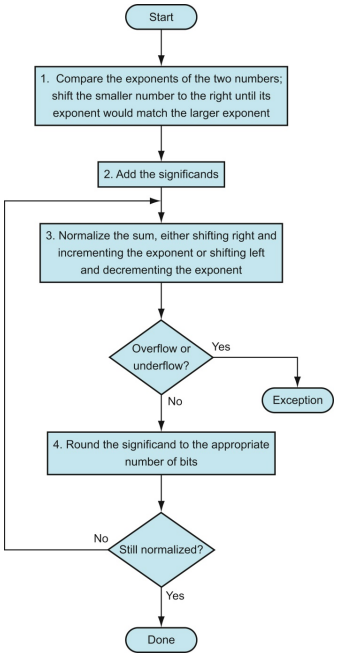
RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
	Set if less than	slt x5, x6, x7	x5 = 1 if x5 < x6, else 0	Compare two registers
	Set if less than, unsigned	sltu x5, x6, x7	x5 = 1 if x5 < x6, else 0	Compare two registers
	Set if less than, immediate	slti x5, x6, x7	x5 = 1 if x5 < x6, else 0	Comparison with immediate
	Set if less than immediate, unsigned	sltiu x5, x6, x7	x5 = 1 if x5 < x6, else 0	Comparison with immediate
	Multiply	mul x5, x6, x7	x5 = x6 × x7	Lower 32 bits of 64-bit product
	Multiply high	mulh x5, x6, x7	x5 = (x6 × x7) >> 32	Upper 32 bits of 64-bit signed product
	Multiply high, unsigned	mulhu x5, x6, x7	x5 = (x6 × x7) >> 32	Upper 32 bits of 64-bit unsigned product
	Multiply high, signed-unsigned	mulhsu x5, x6, x7	x5 = (x6 × x7) >> 32	Upper 32 bits of 64-bit signed-unsigned product
	Divide	div x5, x6, x7	x5 = x6 / x7	Divide signed 32-bit numbers
	Divide unsigned	divu x5, x6, x7	x5 = x6 / x7	Divide unsigned 32-bit numbers
	Remainder	rem x5, x6, x7	x5 = x6 % x7	Remainder of signed 32-bit division
	Remainder unsigned	remu x5, x6, x7	x5 = x6 % x7	Remainder of unsigned 32-bit division
Data transfer	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5, 40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	Uns. byte halfword from memory to register
	Store byte	sb x5, 40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
	Add upper immediate to PC	auipc x5, 0x12345	x5 = PC + 0x12345000	Used for PC-relative data addressing
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >> 3	Arithmetic shift right by immediate
Conditional branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greatr/eq, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
Unconditional branch	Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

Floating point

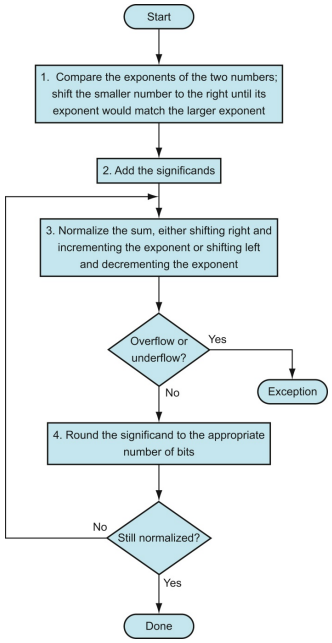
FP Addition

- FP adder hardware
  - much more complex than integer adders
  - usually takes several clock cycles, but can be pipelined
  - designing hardware to perform FP addition in one clock cycle would make it to long, penalizing all other less complex instructions with a slower clock



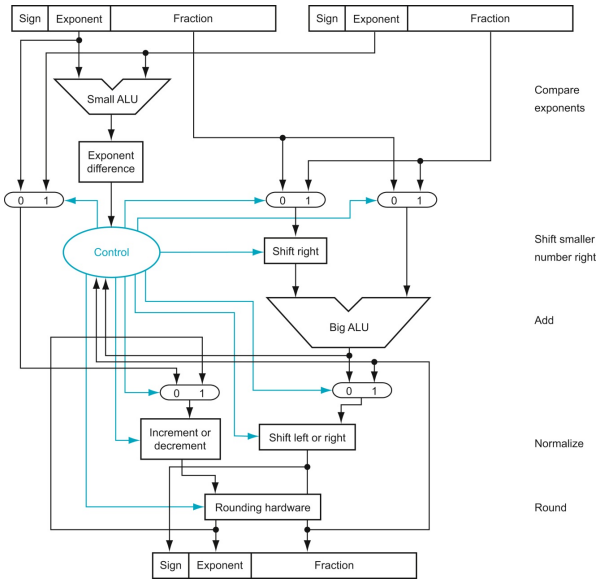
Practice

- $0.5 + -0.4375$
  - binary format
    - $1.000 \times 2^{-1} + -1.110 \times 2^{-2}$
  - shift number with smaller exponent to match the other
    - $1.000 \times 2^{-1} + -0.111 \times 2^{-1}$
  - add significands
    - $1.000 \times 2^{-1} + -0.111 \times 2^{-1} = 0.001 \times 2^{-1}$
  - normalize result and check for overflow/ underflow
    - 1.000×2^{-4}
  - round and renormalize if necessary
    - 1.000×2^{-4} (no change)
    - 0.0625



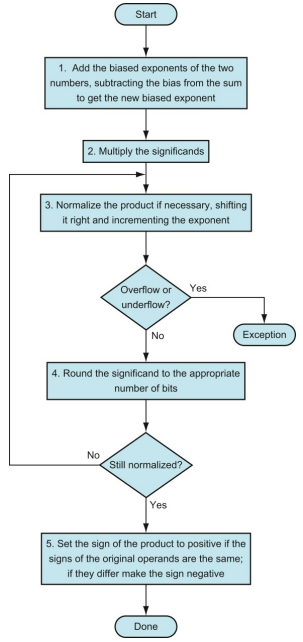
FP Adder

- The exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much;
- This difference controls the three multiplexors; they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU;
- The normalization step then shifts the sum left or right and increments or decrements the exponent;
- Rounding then creates the final result, which may require normalizing again to produce the actual final result.



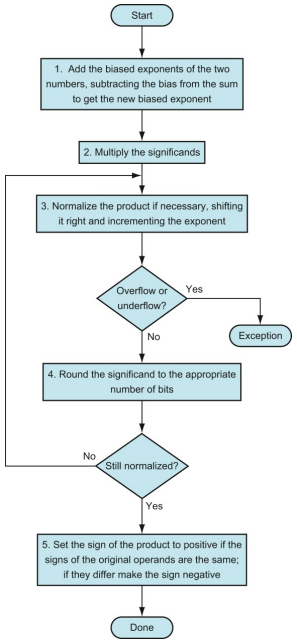
FP Multiplication

- FP multiplier is of similar complexity to FP adder
  - uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually performs:
  - addition, subtraction, multiplication, division, reciprocal, square-root
  - floating point / integer conversions
- Takes several clock cycles, but can be pipelined



Practice

- 0.5×-0.4375
  - binary format
    - $1.000 \times 2^{-1} \times -1.110 \times 2^{-2}$
  - add exponents
    - unbiased: $-1 + -2 = -3$
    - biased: $(-1 + b) + (-2 + b) - b = -3 + b$
  - multiply significands
    - $1.000 \times 1.110 = 1.110 \Rightarrow 1.110 \times 2^{-3}$
  - check result is normalized and check the exponent for overflow/underflow
    - 1.110×2^{-3} (no change)
  - round and renormalize if necessary
    - 1.110×2^{-3} (no change)
  - make the sign negative if necessary
    - -1.110×2^{-3}
    - -0.21875



RISC-V floating-point operands

32 floating-point registers	f0-f31	An *f*-register can hold either a single-precision floating-point number or a double-precision floating-point number.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

RISC-V floating-point assembly language

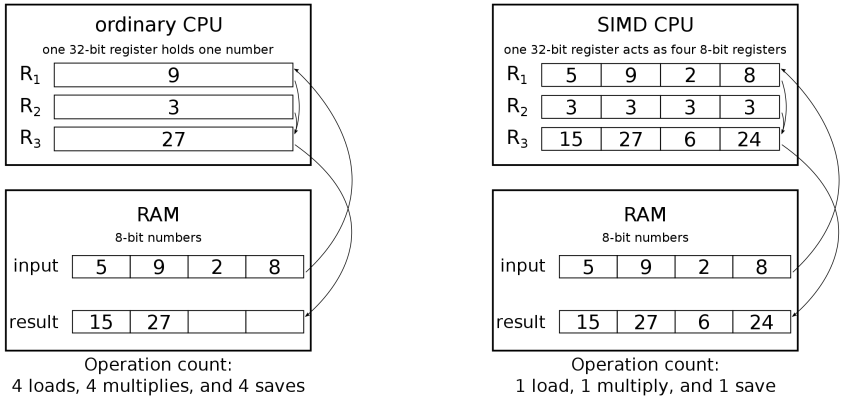
Arithmetic	FP add single	`fadd.s f0, f1, f2`	`f0 = f1 + f2`	FP add (single precision)
	FP subtract single	`fsub.s f0, f1, f2`	`f0 = f1 - f2`	FP subtract (single precision)
	FP multiply single	`fmul.s f0, f1, f2`	`f0 = f1 * f2`	FP multiply (single precision)
	FP divide single	`fdiv.s f0, f1, f2`	`f0 = f1 / f2`	FP divide (single precision)
	FP square root single	`fsqrt.s f0, f1`	`f0 = √f1`	FP square root (single precision)
	FP add double	`fadd.d f0, f1, f2`	`f0 = f1 + f2`	FP add (double precision)
	FP subtract double	`fsub.d f0, f1, f2`	`f0 = f1 - f2`	FP subtract (double precision)
	FP multiply double	`fmul.d f0, f1, f2`	`f0 = f1 * f2`	FP multiply (double precision)
	FP divide double	`fdiv.d f0, f1, f2`	`f0 = f1 / f2`	FP divide (double precision)
	FP square root double	`fsqrt.d f0, f1`	`f0 = √f1`	FP square root (double precision)
Comparison	FP equality single	`feq.s x5, f0, f1`	`x5 = 1 if f0 == f1, else 0`	FP comparison (single precision)
	FP less than single	`flt.s x5, f0, f1`	`x5 = 1 if f0 < f1, else 0`	FP comparison (single precision)
	FP less than or equals single	`fle.s x5, f0, f1`	`x5 = 1 if f0 <= f1, else 0`	FP comparison (single precision)
	FP equality double	`feq.d x5, f0, f1`	`x5 = 1 if f0 == f1, else 0`	FP comparison (double precision)
	FP less than double	`flt.d x5, f0, f1`	`x5 = 1 if f0 < f1, else 0`	FP comparison (double precision)
	FP less than or equals double	`fle.d x5, f0, f1`	`x5 = 1 if f0 <= f1, else 0`	FP comparison (double precision)
Data transfer	FP load word	`flw f0, 4(x5)`	`f0 = Memory[x5 + 4]`	Load single-precision from memory
	FP load doubleword	`fld f0, 8(x5)`	`f0 = Memory[x5 + 8]`	Load double-precision from memory
	FP store word	`fsw f0, 4(x5)`	`Memory[x5 + 4] = f0`	Store single-precision from memory
	FP store doubleword	`fsd f0, 8(x5)`	`Memory[x5 + 8] = f0`	Store double-precision from memory

Arithmetic for multimedia (SIMD)

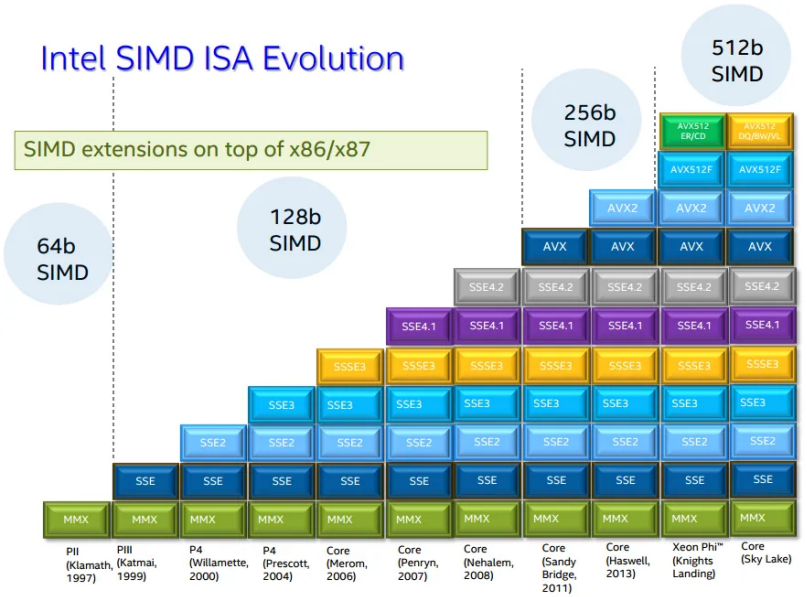
SIMD instructions

- Graphics and media processing operate on values of 8-bit and 16-bit lengths
  - can use a 128-bit adder (with partitioned carry chain) and perform operations in **parallel** — e.g. sixteen 8-bit operations, eight 16-bit operations, or four 32-bit operations
- SIMD **(single-instruction, multiple-data)**
  - a.k.a. data level parallelism, vector parallelism

Tripling four 8-bit numbers



https://en.wikipedia.org/wiki/Single_instruction,_multiple_data



https://en.algorithmica.org/hpc/simd/

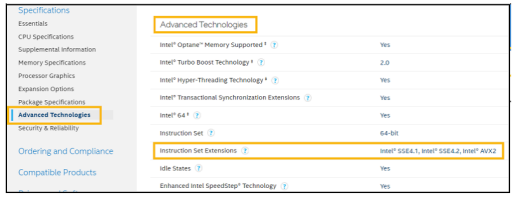
C code

```c
#include <stdio.h>
#include <immintrin.h>

int main() {
    if (__builtin_cpu_is("intel")) printf("Intel CPU\n");
    if (__builtin_cpu_is("amd")) printf("AMD CPU\n");

    if (__builtin_cpu_supports("avx512f")) printf("AVX-512 supported\n");
    if (__builtin_cpu_supports("avx2")) printf("AVX2 supported\n");
    if (__builtin_cpu_supports("avx")) printf("AVX supported\n");
    if (__builtin_cpu_supports("sse2")) printf("SSE2 supported\n");
    if (__builtin_cpu_supports("sse")) printf("SSE supported\n");

    return 0;
}
```

How to check for instruction set extensions?

- **Identify** your Intel® Processor and note the processor number.
- **Go to** the product specification page and **enter** the number of the Intel Processor in the search box.
- **Look** in the *Advanced Technologies* section and **look for** *Instruction Set Extensions*

EXample:



https://www.intel.com/content/www/us/en/support/articles/000090473/processors/intel-core-processors.html