



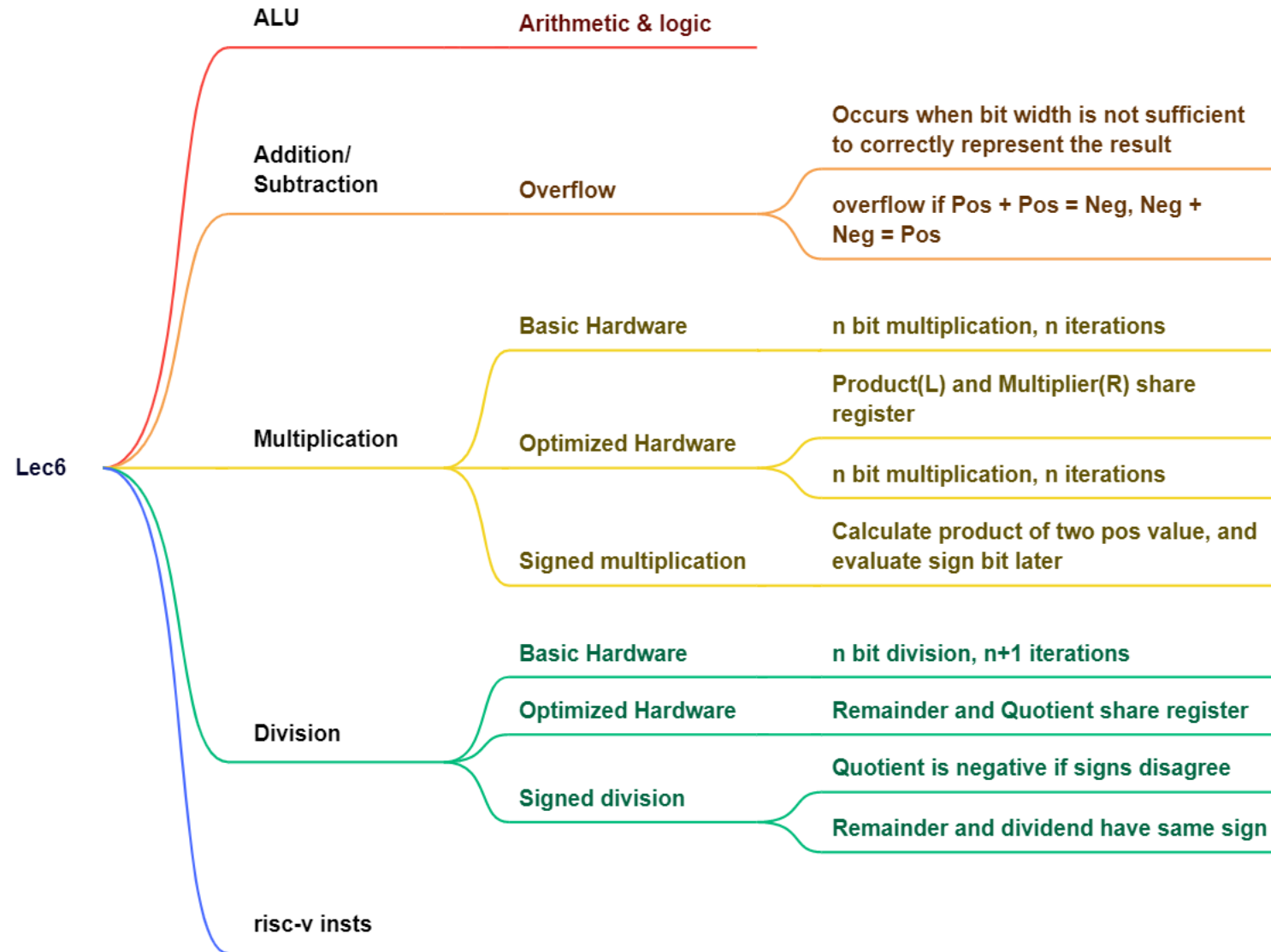
COMPUTER ORGANIZATION

Lecture 7 Floating Point Arithmetic

2025 Spring

This PowerPoint is for internal use only at Southern University of Science and Technology. Please do not repost it on other platforms without permission from the instructor.

Recap



Floating Point

- Recap: Fixed Point v.s. Floating Point
- Representation for non-integral numbers
 - Including very small and very large numbers

- Like scientific notation

- -2.34×10^{56}
- $+0.002 \times 10^{-4}$
- $+987.02 \times 10^9$

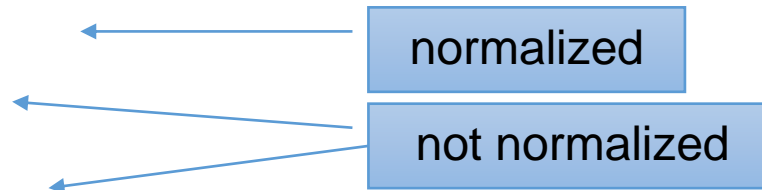
- In binary

- $\pm 1.xxxxxxx_2 \times 2^y$

significand/mantissa

exponent

- Types `float` and `double` in C





Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
 - Portability issues for scientific code
- Now almost universally adopted
- Two representations
 - Single precision (32-bit)
 - Double precision (64-bit)

IEEE Floating-Point Format

$$\pm 1.xxxxxxx_2 \times 2^y$$

single: 8 bits

single: 23 bits

double: 11 bits

double: 52 bits

S	Exponent (y+Bias)	Fraction (xxxx)
---	-------------------	-----------------

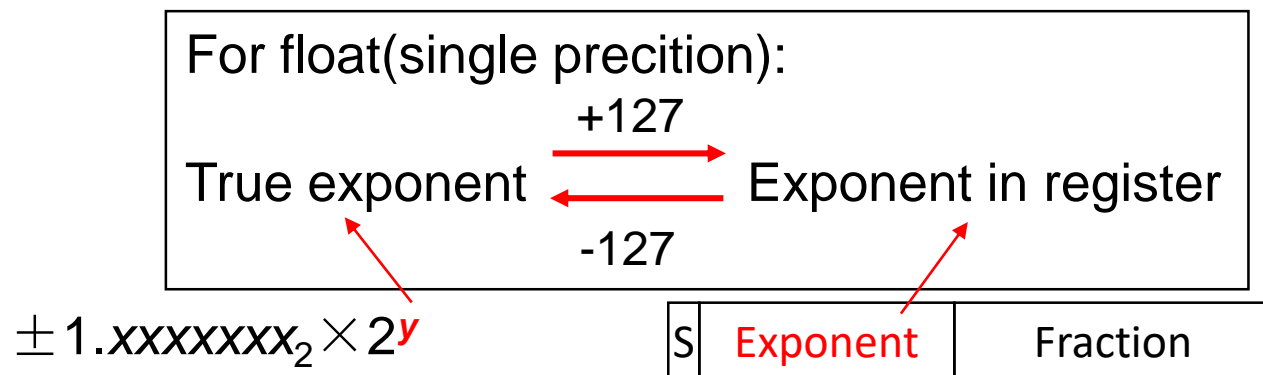
$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 \Rightarrow non-negative, 1 \Rightarrow negative)
- Normalize significand(规约化有效数字)
 - $1.0 \leq |\text{significand}| < 2.0$
 - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (**hidden one**)
 - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent (y) + Bias
 - Ensures exponent is unsigned
 - Single: Bias = 127; Double: Bias = 1023

Discussion: Why we use biased notation in floating point representation?

Bias notation

- Why not use 2's complement for Exponent?
 - Biased exponent is always positive
 - Allows for a more efficient representation of small and large exponents
 - Simplifies the comparison of exponents.



	true exp	biased exp in reg
8 bits	127	254

	0	127
	-1	126

	-126	1
	-127	0
reserved	-128	255

Example 1: Decimal to FP

- Represent -0.75
 - $-0.75_{\text{ten}} = (-1)^1 \times 1.1_2 \times 2^{-1}$
 - $S = 1$
 - Fraction = $1000\dots00_2$
 - Exponent = $-1 + \text{Bias}$
 - Single: $-1 + 127 = 126 = 01111110_2$
 - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: $1_01111110_1000\dots00$ 23bits
- Double: $1_011111111110_1000\dots00$ 52bits

Exercise: Represent 24.5_{ten} in single-precision FP

- Sign bit = ?
- Fraction = ?
- Exponent = ?_{ten}

Recall:

True exponent $\xrightarrow{+127}$ Exponent in register
 $\xleftarrow{-127}$

Example 2: FP to decimal

- What number is represented by the single-precision float

11000000101000...00

- $S = 1$
 - Fraction = 01000...00₂
 - Exponent = 10000001₂ = 129
- $x = (-1)^1 \times (1.01_2) \times 2^{(129 - 127)}$
 $= (-1) \times 1.25_{\text{ten}} \times 2^2$
 $= -5.0_{\text{ten}}$

Exercise: Represent
single-precision FP to
decimal

0_10000011_1100...00

Recall:

True exponent $\xrightarrow{+127}$ Exponent in register
 $\xleftarrow{-127}$

Single-Precision Range

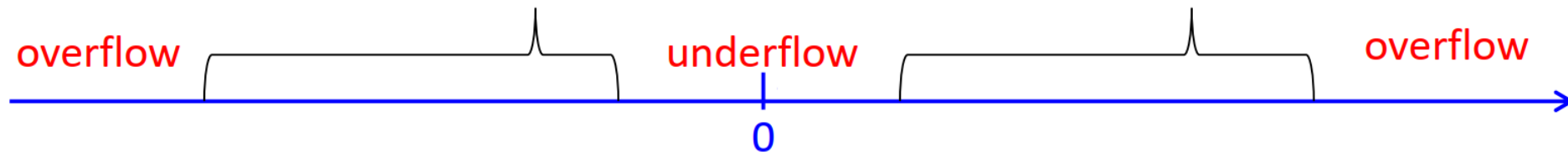
- Exponents 00000000 and 11111111 are reserved
- Smallest value
 - Exponent: 00000001 \Rightarrow actual exponent = $1 - 127 = -126$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-126} (\approx \pm 1.2 \times 10^{-38})$
- Largest value
 - exponent: 11111110 \Rightarrow actual exponent = $254 - 127 = +127$
 - Fraction: 111...11 \Rightarrow significand ≈ 2.0
 - $\pm 2.0 \times 2^{+127} (\approx \pm 3.4 \times 10^{+38})$
- Range: $(-2.0 \times 2^{127}, -1.0 \times 2^{-126}], [1.0 \times 2^{-126}, 2.0 \times 2^{127})$

Double-Precision Range

- Exponents 0000...00 and 1111...11 are reserved
- Smallest value
 - Exponent: 00000000001 \Rightarrow actual exponent = $1 - 1023 = -1022$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-1022} (\approx \pm 2.2 \times 10^{-308})$
- Largest value
 - Exponent: 11111111110 \Rightarrow actual exponent = $2046 - 1023 = 1023$
 - Fraction: 111...11 \Rightarrow significand ≈ 2.0
 - $\pm 2.0 \times 2^{+1023} (\approx \pm 1.8 \times 10^{+308})$
- Range: $(-2.0 \times 2^{1023}, -1.0 \times 2^{-1022}], [1.0 \times 2^{-1022}, 2.0 \times 2^{1023})$

Overflow and Underflow

- Range of float: $(-2.0 \times 2^{127}, -1.0 \times 2^{-126}], [1.0 \times 2^{-126}, 2.0 \times 2^{127})$
- Range of double: $(-2.0 \times 2^{1023}, -1.0 \times 2^{-1022}], [1.0 \times 2^{-1022}, 2.0 \times 2^{1023})$



- **Overflow**: when the exponent is **too large** to be represented
- **Underflow**: when is **negative** exponent is **too large** to be represented (when it's exponent is too small to be represented)
- Examples:
 - For float number, 8-bit exponent, range: -126~127
 - 1×2^{128} , -1.1×2^{129} **Overflow**
 - 1×2^{-127} , -1.1×2^{-128} **Underflow**
 - For double number, 11-bit exponent, range: -1022~1023
 - 1×2^{1024} , -1.1×2^{1026} **Overflow**
 - 1×2^{-1023} , -1.1×2^{-1025} **Underflow**

IEEE 754 Encoding of FPN

- ± 0 , $\pm \infty$ (infinity), NaN

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1–254	Anything	1–2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

- $\pm \infty$: divided by 0
- NaN : 0/0, subtracting infinity from infinity

Gradual Underflow

- Represent denormalized numbers
 - Exponent : all zeros
 - Significand : non-zeros (but no hidden one)
 - Allow a number to degrade in significance until it become 0 (gradual underflow)
- The smallest **normalized** number (significand has hidden one)
 - $1.0000\ 0000\ 0000\ 0000\ 0000\ 000 \times 2^{-126}$
- The smallest **de-normalized** number (significand has no hidden one any more)
 - $0.0000\ 0000\ 0000\ 0000\ 0000\ 001 \times 2^{-126}$

Infinites and NaNs

- Exponent = 111...1, Fraction = 000...0
 - \pm Infinity
 - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction \neq 000...0
 - Not-a-Number (NaN)
 - Indicates illegal or undefined result
 - e.g., 0.0 / 0.0
 - Can be used in subsequent calculations

Floating-Point Precision

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent (=y+Bias)	Fraction (xxxx)
---	--------------------	-----------------

- Relative precision

- all fraction bits are significant
- assuming $a = 1.00\dots0 \times 2^y$, $b = 1.00\dots1 \times 2^y$

$$\bullet \frac{a-b}{a} = \frac{\Delta A}{|A|} = \frac{2^{-23} \times 2^y}{|1.\text{xxx} \times 2^y|} \leq \frac{2^{-23} \times 2^y}{1 \times 2^y} = 2^{-23}$$

- Single: approx 2^{-23}
 - Equivalent to $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
- Double: approx 2^{-52}
 - Equivalent to $52 \times \log_{10} 2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Addition Example 1: decimal

- Consider a 4-digit decimal example
 - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
 - Shift number with smaller exponent
 - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. Add significands
 - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- 3. Normalize result & check for over/underflow
 - 1.0015×10^2
- 4. Round and renormalize if necessary
 - 1.002×10^2

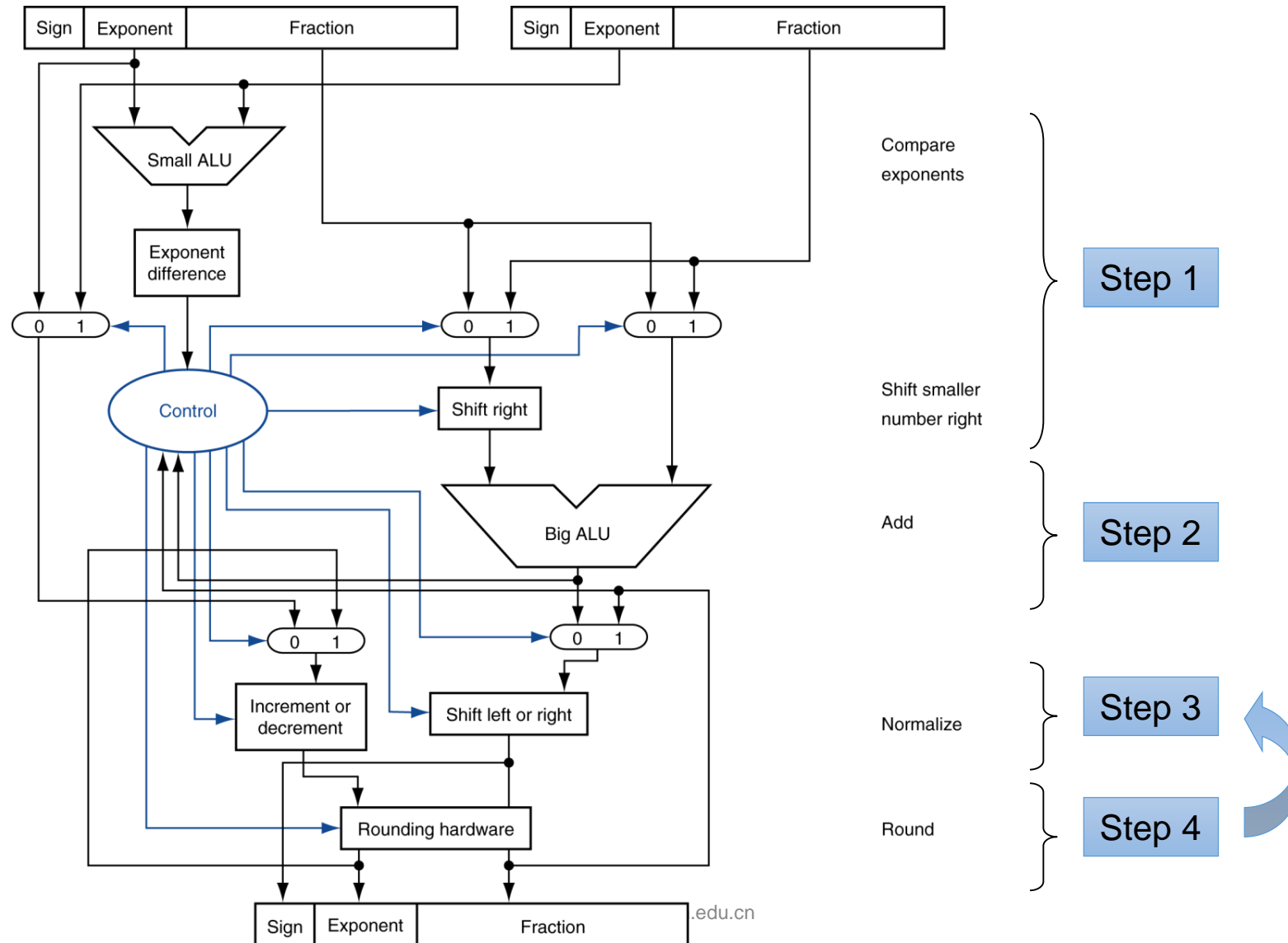
Addition Example 2: FP

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + -0.4375)
- 1. Align binary points
 - Shift number with smaller exponent
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
 - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
 - $1.000_2 \times 2^{-4}$ (no change) = 0.0625₁₀

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
 - Much longer than integer operations
 - Slower clock would penalize all instructions
- FP adder usually takes several cycles
 - Can be pipelined

FP Adder Hardware



0.5 - 0.4375 preserve 4digit

$$0.5 = 1.000_2 \times 2^{-1}$$

$$S = 0$$

$$\text{Exp.} = -1 + 127 = 126$$

$$\text{Frac.} = 0000...00_2$$

$$-0.4375 = -1.110_2 \times 2^{-2}$$

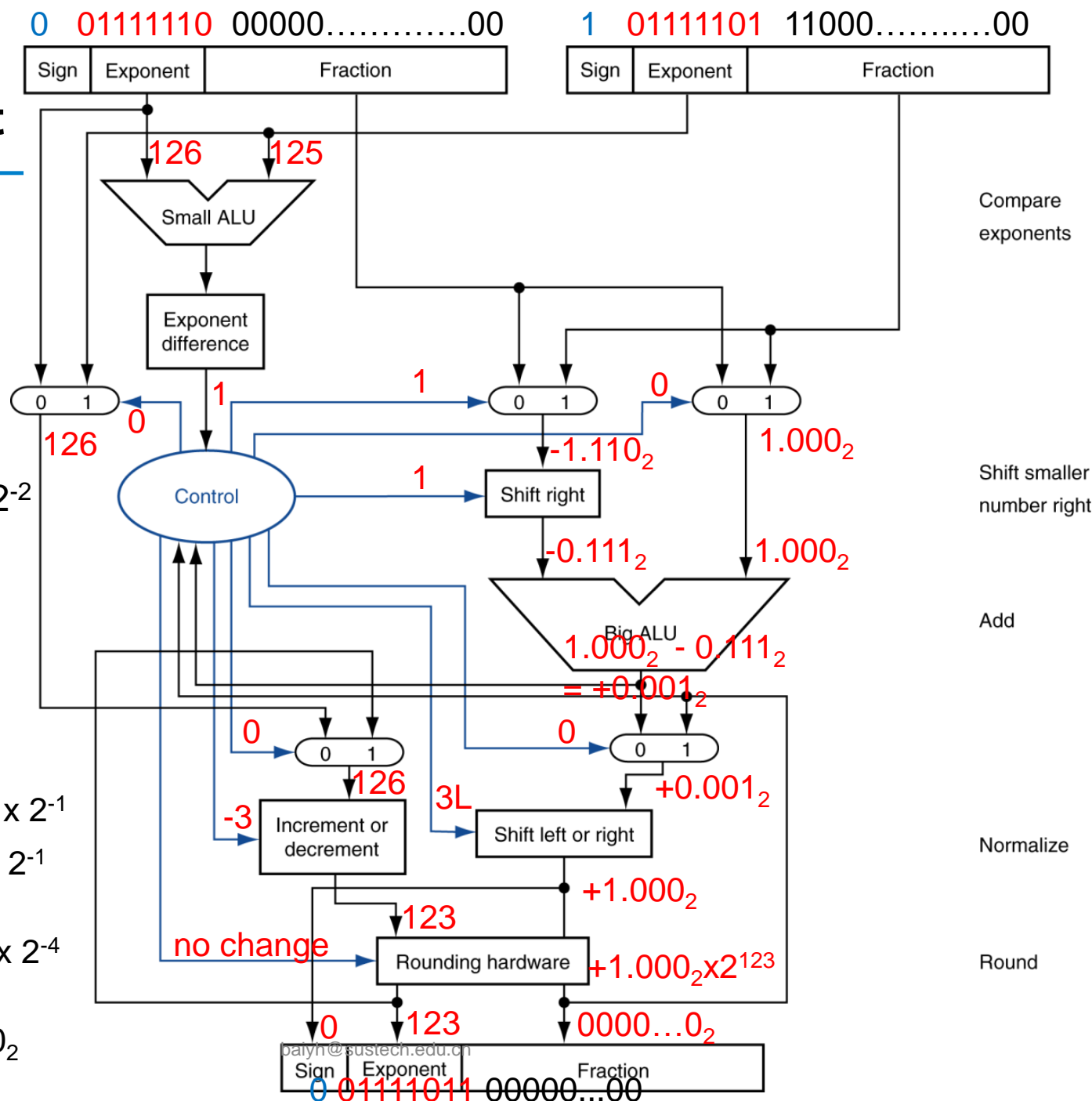
$$S = 1$$

$$\text{Exp.} = -2 + 127 = 125$$

$$\text{Frac.} = 1100...00_2$$

steps:

- $-1.110_2 \times 2^2 = -0.111_2 \times 2^{-1}$
- $1.000_2 \times 2^{-1} - 0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- $0.001_2 \times 2^{-1} = 1.000_2 \times 2^{-4}$
- $1.000_2 \times 2^{-4} =$
 $00111101100000...00_2$



FP Multiplication

- Similar steps
 - Compute exponent (careful!)
 - Multiply significands (set the binary point correctly)
 - Normalize
 - Round (potentially re-normalize)
 - Assign sign

Multiplication Example 1: decimal

- Consider a 4-digit decimal example
 - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
 - For biased exponents, subtract bias from sum
 - New exponent = $10 + -5 = 5$
- 2. Multiply significands
 - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$
- 3. Normalize result & check for over/underflow
 - 1.0212×10^6
- 4. Round and renormalize if necessary
 - 1.021×10^6
- 5. Determine sign of result from signs of operands
 - $+1.021 \times 10^6$

Multiplication Example 1: FP

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
 - Unbiased: $-1 + -2 = -3$
 - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
 - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
 - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
 - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: $+ve \times -ve \Rightarrow -ve$
 - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
 - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
 - Addition, subtraction, multiplication, division, reciprocal, square-root
 - FP \leftrightarrow integer conversion
- Operations usually takes several cycles
 - Can be pipelined

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
 - Extra bits of precision (guard, round, sticky)
 - 1st bit: **Guard** bit (5), 2nd bit: **Round** bit (6), 3rd bit: **Sticky** bit
 - 2.56+234 , 237 vs. 236 (assume that we have three significant digits)

$$\begin{array}{r} 2.3400_{\text{ten}} \\ + 0.0256_{\text{ten}} \\ \hline 2.3656_{\text{ten}} \end{array} \quad \begin{array}{r} 2.34_{\text{ten}} \\ + 0.02_{\text{ten}} \\ \hline 2.36_{\text{ten}} \end{array}$$

- IEEE 754 guarantee one-half (0.5) ulp (units in the last place)
- Choice of rounding modes
 - Round up, round down, truncate, **round to the nearest even** (for X.50)
 - To the nearest even: (binary number $\underline{0}.10 \rightarrow 0$, $\underline{1}.10 \rightarrow 10$)
- Allows programmer to fine-tune numerical behavior of a computation
- Trade-off between hardware complexity, performance, and market requirements

FP Instructions in RISC-V

- Separate FP registers: f0, ..., f31
 - double-precision
 - single-precision values stored in the lower 32 bits
- FP instructions operate only on FP registers
 - Programs generally don't do integer ops on FP data, or vice versa
 - More registers with minimal code-size impact
- FP load and store instructions
 - flw, fsw
 - fld, fsd

FP Instructions in RISC-V

- Single-precision arithmetic
 - `fadd.s`, `fsub.s`, `fmul.s`, `fdiv.s`, `fsqrt.s`
 - e.g., `fadds.s f2, f4, f6`
- Double-precision arithmetic
 - `fadd.d`, `fsub.d`, `fmul.d`, `fdiv.d`, `fsqrt.d`
 - e.g., `fadd.d f2, f4, f6`
- Single- and double-precision comparison
 - `feq.s`, `flt.s`, `fle.s`
 - `feq.d`, `flt.d`, `fle.d`
 - Result is 0 or 1 in integer destination register
 - Use `beq`, `bne` to branch on comparison result
- Branch on FP condition code true or false
 - `B.cond`

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {  
    return ((5.0/9.0)*(fahr - 32.0));  
}
```

- fahr in f10, result in f10, literals in global memory space

- Compiled RISC-V code:

f2c:

```
f1w    f0,const5(x3)    # f0 = 5.0f  
f1w    f1,const9(x3)    # f1 = 9.0f  
fdiv.s f0, f0, f1       # f0 = 5.0f / 9.0f  
f1w    f1,const32(x3)   # f1 = 32.0f  
fsub.s f10,f10,f1       # f10 = fahr - 32.0  
fmul.s f10,f0,f10       # f10 = (5.0f/9.0f) * (fahr-32.0f)  
jalr   x0,0(x1)        # return
```

const5,const9,const32 stands for the value's corresponding offset

FP Example: Array Multiplication

- $X = X + Y \times Z$
 - All 32×32 matrices, 64-bit double precision elements

- C code:

```
void mm (double c[][],  
         double a[][], double b[][]) {  
    int i, j, k;  
    for (i = 0; i != 32; i = i + 1)  
        for (j = 0; j != 32; j = j + 1)  
            for (k = 0; k != 32; k = k + 1)  
                c[i][j] = c[i][j]  
                    + a[i][k] * b[k][j];  
}
```

- Addresses of c, a, b in x10, x11, x12, and
i, j, k in x5, x6, x7

RISC-V
code:

	li	x28, 32	# x28 = 32 (row size/loop end)
	li	x5, 0	# i = 0; initialize 1st for loop
L1:	li	x6, 0	# j = 0; initialize 2nd for loop
L2:	li	x7, 0	# k = 0; initialize 3rd for loop
	slli	x30, x5, 5	# x30 = i * 2 ⁵ (size of row of c)
	add	x30, x30, x6	# x30 = i * size(row) + j
	slli	x30, x30, 3	# x30 = byte offset of [i][j]
	add	x30, x10, x30	# x30 = byte address of c[i][j]
	fld	f0, 0(x30)	# f0 = c[i][j]
L3:	slli	x29, x7, 5	# x29 = k * 2 ⁵ (size of row of b)
	add	x29, x29, x6	# x29 = k * size(row) + j
	slli	x29, x29, 3	# x29 = byte offset of [k][j]
	add	x29, x12, x29	# x29 = byte address of b[k][j]
	fld	f1, 0(x29)	# f1 = b[k][j]
	slli	x29, x5, 5	# x29 = i * 2 ⁵ (size of row of a)
	add	x29, x29, x7	# x29 = i * size(row) + k
	slli	x29, x29, 3	# x29 = byte offset of [i][k]
	add	x29, x11, x29	# x29 = byte address of a[i][k]
	fld	f2, 0(x29)	# f2 = a[i][k]
	fmul.d	f1, f2, f1	# f1 = a[i][k] * b[k][j]
	fadd.d	f0, f0, f1	# f0 = c[i][j] + a[i][k] * b[k][j]
	addi	x7, x7, 1	# k = k + 1
	bltu	x7, x28, L3	# if (k < 32) go to L3
	fsd	f0, 0(x30)	# c[i][j] = f0
	addi	x6, x6, 1	# j = j + 1
	bltu	x6, x28, L2	# if (j < 32) go to L2
	addi	x5, x5, 1	# i = i + 1
	bltu	x5, x28, L1	# if (i < 32) go to L1

Subword Parallelism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
 - Example: 128-bit adder:
 - Sixteen 8-bit adds
 - Eight 16-bit adds
 - Four 32-bit adds
- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)

Streaming SIMD Extension 2 (SSE2)

- Adds 4×128 -bit registers
 - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
 - 2×64 -bit double precision
 - 4×32 -bit double precision
 - Instructions operate on them simultaneously
 - Single-Instruction Multiple-Data

Matrix Multiply

- Unoptimized code:

```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.   for (int i = 0; i < n; ++i)
4.     for (int j = 0; j < n; ++j)
5.       {
6.         double cij = C[i+j*n]; /* cij = C[i][j] */
7.         for(int k = 0; k < n; k++ )
8.           cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.         C[i+j*n] = cij; /* C[i][j] = cij */
10.      }
11. }
```



Matrix Multiply

• x86 assembly code:

```
1. vmovsd (%r10),%xmm0    # Load 1 element of C into %xmm0
2. mov %rsi,%rcx          # register %rcx = %rsi
3. xor %eax,%eax          # register %eax = 0
4. vmovsd (%rcx),%xmm1    # Load 1 element of B into %xmm1
5. add %r9,%rcx           # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1, element of A
7. add $0x1,%rax          # register %rax = %rax + 1
8. cmp %eax,%edi          # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
10. jg 30 <dgemm+0x30>    # jump if %eax > %edi
11. add $0x1,%r11d        # register %r11 = %r11 + 1
12. vmovsd %xmm0, (%r10)  # Store %xmm0 into C element
```



Matrix Multiply

- Optimized C code:

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.     for ( int i = 0; i < n; i+=4 )
5.         for ( int j = 0; j < n; j++ ) {
6.             __m256d c0 = __mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.             for( int k = 0; k < n; k++ )
8.                 c0 = __mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.                                     __mm256_mul_pd(__mm256_load_pd(A+i+k*n),
10.                                                     __mm256_broadcast_sd(B+k+j*n)));
11.             __mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.         }
13. }
```

Matrix Multiply

• Optimized x86 assembly code:

```
1. vmovapd (%r11),%ymm0      # Load 4 elements of C into %ymm0
2. mov %rbx,%rcx             # register %rcx = %rbx
3. xor %eax,%eax             # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5. add $0x8,%rax             # register %rax = %rax + 8
6. vmulpd (%rcx),%ymm1,%ymm1 # Parallel mul %ymm1, 4 A elements
7. add %r9,%rcx              # register %rcx = %rcx + %r9
8. cmp %r10,%rax             # compare %r10 to %rax
9. vaddpd %ymm1,%ymm0,%ymm0  # Parallel add %ymm1, %ymm0
10. jne 50 <dgemm+0x50>      # jump if not %r10 != %rax
11. add $0x1,%esi            # register % esi = % esi + 1
12. vmovapd %ymm0, (%r11)    # Store %ymm0 into 4 C elements
```

Concluding Remarks

- Bits have no inherent meaning
 - Interpretation depends on the instructions applied
- Computer representations of numbers
 - Finite range and precision
 - Need to account for this in programs
- ISAs support arithmetic
 - Signed and unsigned integers
 - Floating-point approximation to reals
- Bounded range and precision
 - Operations can overflow and underflow