Arithmetic for Computers

Binary Representation

Unsigned :

- The binary number

01011000 00010101 00101110 11100111

MSB LSB

represents the quantity $0 \times 2^{31} + 1 \times 2^{30} + \cdots + 1 \times 2^{0}$

- A 32-bit word can represent 2^{32} numbers between 0 and $2^{32} - 1$ Numbers are always positive

Binary Representation

2's Complement:

32 bits can only represent 2^{32} numbers – if we wish to also represent negative numbers, we can represent 2^{31} positive numbers (incl zero) and 2^{31} negative numbers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}\text{-}1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} – 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} – 2)$
…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

Each number represents the quantity
$-2^{31}\ x_{31} + x_{30}\ 2^{30} + x_{29}\ 2^{29} + … + x_1\ 2^1 + x_0\ 2^0$

Why is this representation favorable?
Consider the sum of 1 and -2 …. we get -1
Consider the sum of 2 and -1 …. we get +1
This format can directly undergo addition without any conversions!

Binary Representation

2's Complement:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}\text{-}1$

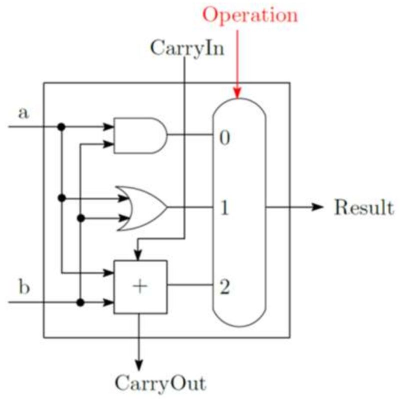
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} – 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} – 2)$
…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

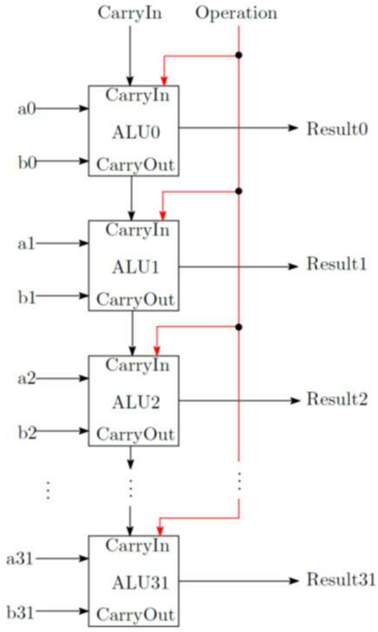
Note that the sum of a number x and its inverted representation x' always equals a string of 1s (-1)
$x + x' = -1$
$x' + 1 = -x$ … hence, can compute the negative of a number by
$-x = x' + 1$ inverting all bits and adding 1

Similarly, the sum of x and -x gives us all zeroes, with a carry of 1
In reality, $x + (-x) = 2^n$ … hence the name 2's complement

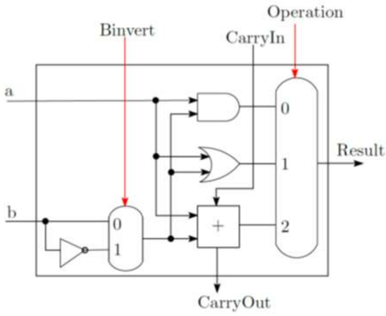
ALU Design

1bit ALU:





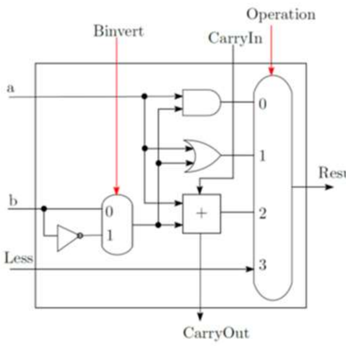
Q. How to do subtraction?



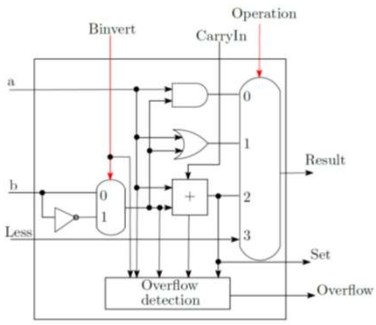
Q. How to design 32-bit ALU from 1-bit ALU?

Tailoring the 32-bit ALU to MIPS

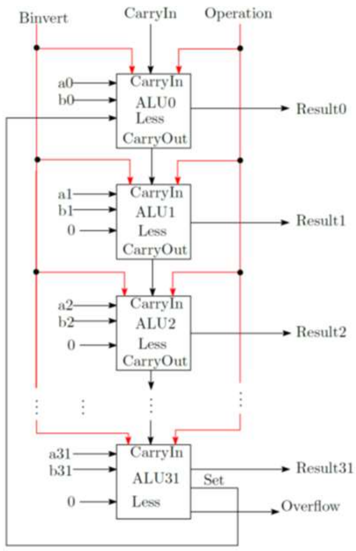
slt rd, rs, rt -- if (rs < rt) rd = 1; else rd = 0 **Idea:** (a-b) < 0 → a < b



ALU 0 – ALU 30



ALU 31



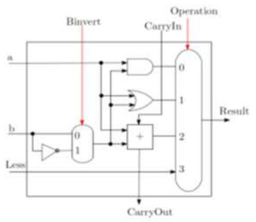
Problem: Overflow e.g., rs = $(1001)_2$ = -7 and rt = $(0110)_2$= 6

How to modify the 1-bit ALU to handle slt correctly?

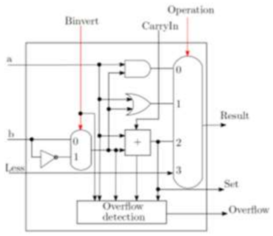
Tailoring the 32-bit ALU to MIPS

Supporting conditional branch: beq rs, rt, L1 -- if (rs == rt) branch to instruction labeled L1

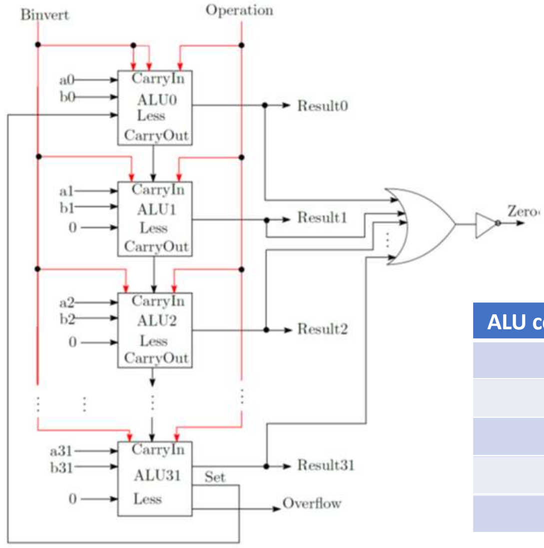
Idea: (a-b) = 0 → a = b

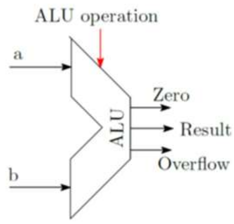


ALU 0 – ALU 30



ALU 31

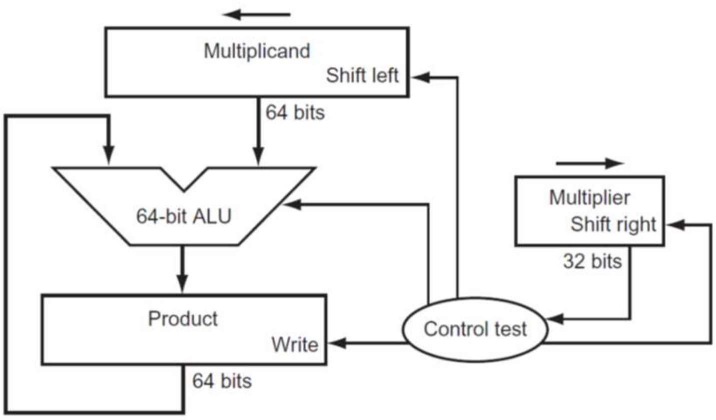




ALU control lines	Function
000	and
001	or
010	add
110	subtract
111	Set on less than

Unsigned Multiplication

```
      1 1 0 1   Multiplicand (13)
      1 0 1 1   Multiplier (11)
      ---------
      1 1 0 1
    1 1 0 1
  0 0 0 0
1 1 0 1
-----------------
1 0 0 0 1 1 1 1   Product (143)
```

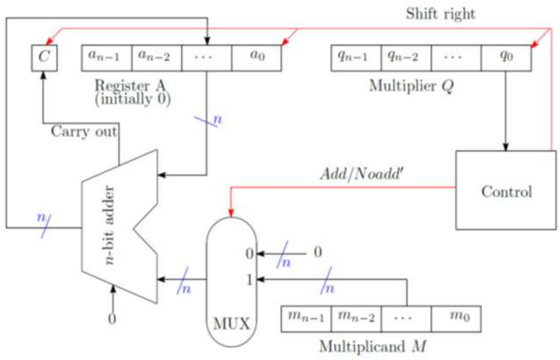


Unsigned Multiplication

Unsigned multiplication:

M: 1 1 0 1

C	A	Q	
0	0 0 0 0	1 0 1 1	Initialization
0	1 1 0 1	1 0 1 1	q0 = 1 → Add
0	0 1 1 0	1 1 0 1	Shift right
1	0 0 1 1	1 0 1 1	q0 = 1 → Add
0	1 0 0 1	1 1 1 0	Shift right
0	1 0 0 1	1 1 1 0	q0 = 0 → No Add
0	0 1 0 0	1 1 1 1	Shift right
1	0 0 0 1	1 1 1 1	q0 = 1 → Add
0	1 0 0 0	1 1 1 1	Shift right




```
        1 1 0 1   Multiplicand (13)
        1 0 1 1   Multiplier (11)
        ---------
        1 1 0 1
      1 1 0 1
    0 0 0 0
  1 1 0 1
-----------------
1 0 0 0 1 1 1 1   Product (143)
```

Signed Multiplication

- Recall grade school trick
  - When multiplying by 9:
    - Multiply by 10 (shift digits left)
    - Subtract once
    - E.g., 12345 x 9 = 12345 x (10 - 1)
      = 123450 - 12345
- Booth's algorithm applies same principle
  - Except no '9' in binary, just '1' and '0'
- Search for a run of '1' bits in the multiplier
  - E.g. '0110' has a run of 2 '1' bits in the middle
  - Multiplying by '0110' (6 in decimal) is equivalent to multiplying by 8 and subtracting twice, since 6 x m = (8 – 2) x m = 8m – 2m

Booth's encoding:

Current bit	Bit to right	Explanation	Example	Operation
1	0	Begins run of '1'	0000111100	Subtract
1	1	Middle of run of '1'	0000111100	Nothing
0	1	End of a run of '1'	0000111100	Add
0	0	Middle of a run of '0'	0000111100	Nothing

Booth's Algorithm

Example: 2 x -3 = -6, or 0010 x 1101 = 1111 1010

M: 0010

A	Q	Q_{-1}	
0000	1101	0	Initialization
1110	1101	0	Q_0Q_{-1} = 10 → Subtract
1111	0110	1	Arithmetic right shift
0001	0110	1	Q_0Q_{-1} = 01 → Add
0000	1011	0	Arithmetic right shift
1110	1011	0	Q_0Q_{-1} = 10 → Subtract
1111	0101	1	Arithmetic right shift
1111	0101	1	Q_0Q_{-1} = 11 → Nothing
1111	1010	1	Arithmetic right shift

Current bit	Bit to right	Operation
1	0	Subtract
1	1	Nothing
0	1	Add
0	0	Nothing

Why Booth's Algorithm Works?

Assume a be the multiplier and b be the multiplicand

Booth's algorithm can be written as

$$(a_{-1} - a_0) \times b \times 2^0$$
$$+ (a_0 - a_1) \times b \times 2^1$$
$$+ (a_1 - a_2) \times b \times 2^2$$
$$\ldots \quad \ldots$$
$$+ (a_{29} - a_{30}) \times b \times 2^{30}$$
$$+ (a_{30} - a_{31}) \times b \times 2^{31}$$

Note: $-a_i \times 2^i + a_i \times 2^{i+1} = a_i \times 2^i$ and $a_{-1} = 0$

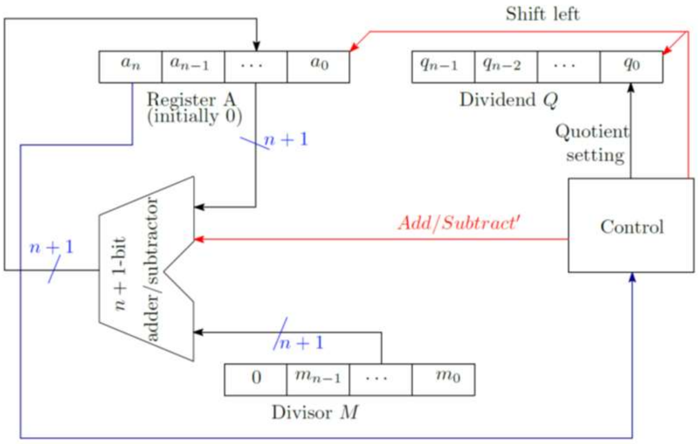
a_i	a_{i-1}	Operation
1	0	Subtract
1	1	Nothing
0	1	Add
0	0	Nothing

$a_{i-1} - a_i = 0$ Nothing
$a_{i-1} - a_i = -1$ Add
$a_{i-1} - a_i = 1$ Subtract

Factoring out b from each term: $b \times \Big(\underbrace{(a_{31} \times -2^{31}) + (a_{30} \times 2^{30}) + \cdots + (a_1 \times 2^1) + (a_0 \times 2^0)}_{\text{2's complement of } a}\Big)$

Unsigned Division

```
1101   | 100010010   Dividend
Divisor  1101
          10000
           1101
             1110
             1101
                1 Remainder
```



After division n-bit quotient is in Q and the remainder is in A

Restoring division:

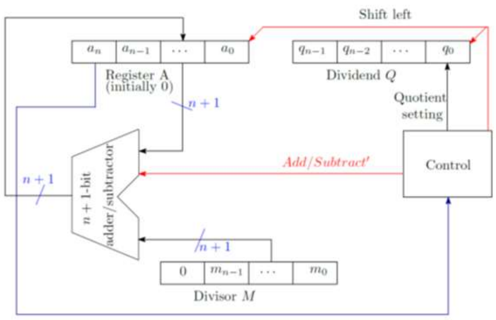
Do the following n times

1. Shift A and Q left one bit
2. $A \leftarrow A - M$
3. **If** the sign of A is 1 i.e., $a_n = 1$
   $q_0 \leftarrow 0$
   $A \leftarrow A + M$ //Restore
   else
   $q_0 \leftarrow 1$

Unsigned Division

Example: Divisor: 11 Dividend: 1000

A	Q	M	
0 0 0 0 0	1 0 0 0	0 0 0 1 1	Initialization
0 0 0 0 1	0 0 0 ?		Shift left AQ
1 1 1 0 1	0 0 0 ?		Subtract $A \leftarrow A - M$
0 0 0 0 1	0 0 0 **0**		$q_0 \leftarrow 0, A \leftarrow A + M$
0 0 0 1 0	0 0 0 ?		Shift left AQ
1 1 1 1 1	0 0 0 ?		Subtract $A \leftarrow A - M$
0 0 0 1 0	0 0 0 **0**		$q_0 \leftarrow 0, A \leftarrow A + M$
0 0 1 0 0	0 0 0 ?		Shift left AQ
0 0 0 0 1	0 0 0 ?		Subtract $A \leftarrow A - M$
0 0 0 0 1	0 0 0 **1**		$q_0 \leftarrow 1$
0 0 0 1 0	0 0 1 ?		Shift left AQ
1 1 1 1 1	0 0 1 ?		Subtract $A \leftarrow A - M$
0 0 0 1 0	0 0 1 **0**		$q_0 \leftarrow 0, A \leftarrow A + M$
Remainder	Quotient		



Do the following n times

1. Shift A and Q left one bit
2. $A \leftarrow A - M$
3. **If** the sign of A is 1 i.e., $a_n = 1$
   $q_0 \leftarrow 0$
   $A \leftarrow A + M$ //Restore
   else
   $q_0 \leftarrow 1$

Signed Division

Simplest solution: Use unsigned division and negate the quotient if signs of divisor and dividend disagree.

Q. What should be the sign of remainder?

Note: The equation $Dividend = Quotient \times Divisor + Remainder$ must always hold

Example: $-7 \div 2$: $Quotient = -3$,
$Remainder = Dividend - Quotient \times Divisor = -7 - (-3 \times 2) = -1$

Note: -4 as quotient also satisfies the formula i.e., $Remainder = -7 - (-4 \times 2) = 1$.
However, the absolute value of the quotient would change depending on the sign of the divisor and the dividend. Here, $-(x \div y) \neq (-x) \div y$, which is problematic!

Example: $7 \div -2$: $Quotient = -3$,
$Remainder = Dividend - Quotient \times Divisor = 7 - (-3 \times -2) = 1$

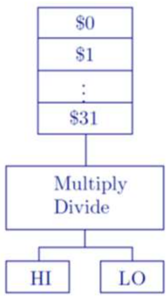
Example: $-7 \div -2$: $Quotient = 3$,
$Remainder = Dividend - Quotient \times Divisor = -7 - (3 \times -2) = -1$

Rule: Negate the quotient if the sign of divisor and dividend are opposite.
Sign of non-zero remainder matches the dividend.

MIPS Multiplication and Division

Multiplication:

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- **Instructions**
  - mult rs, rt / multu rs, rt
    - 64-bit product in HI/LO
  - mfhi rd / mflo rd
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits



Division:

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- **Instructions**
  - div rs, rt / divu rs, rt
  - No overflow or divide-by-0 checking
    - If divisor is 0, result is unpredictable
    - Software must perform checks if required
  - Use mfhi, mflo to access result

Floating Point Numbers

Normalized scientific notation: single non-zero digit to the left of the decimal (binary) point
– example: 3.5×10^9

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

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

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

- S: sign bit ($0 \Rightarrow$ non-negative, $1 \Rightarrow$ negative)
- **Normalize significand:** $1.0 \leq |\text{significand}| < 2.0$
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- **Exponent:** excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Q. Why excess-127 representation for exponent is used?

Note: More exponent bits → wider range of numbers
More fraction bits → higher precision

Example

- Represent 0.75
  - $0.75 = 1.1_2 \times 2^{-1}$
  - S = 0
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
- Single: 0011111101000…00

Q. Represent 0.5

Q. How to represent 0 ?

A. Exponent 00…0, Fraction: 00…0

±Infinity: Exponent = 111...1, Fraction = 000...0

NAN: Exponent = 111...1, Fraction ≠ 000...0
Indicates illegal or undefined result e.g., 0.0/0.0

- What number is represented by the single-precision float

11000000101000…00
  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = 10000001_2 = 129

- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129-127)}$
  $= (-1) \times 1.25 \times 2^2$
  $= -5.0$

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

Floating Point Numbers

Single Precision Range:

- Exponents 00000000 and 11111111 reserved

Smallest value

Exponent: 00000001
⇒ actual exponent = 1 – 127 = –126
Fraction: 000…00 ⇒ significand = 1.0
$\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

Largest value

exponent: 11111110
⇒ actual exponent = 254 – 127 = +127
Fraction: 111…11 ⇒ significand ≈ 2.0
$\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double Precision Range:

- Exponents 0000…00 and 1111…11 reserved
- **Smallest value**
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- **Largest value**
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Summary

Value inf — 2 special cases up top that use the reserved exponent field of 255 — 0 255 00…0

Value NAN — 0 255 xx….x

Highest Norm ~2 x 2^{127} — 0 254 11….1

Value 1 — 0 127 00…0

Exponent field < 127, i.e., after subtracting bias, they are negative exponents, representing numbers < 1

Smallest Norm ~1x 2^{-126} — 0 0..01 00…0

Largest Denorm ~1 x 2^{-126} — 0 0..00 11…1

Smallest Denorm ~2^{-149} — 0 0..00 00…1

Special case with exponent field 0, used to represent denorms, that help us gradually approach 0

Value 0 — 0 0..00 00…0

Same rules as above, but the sign bit is 1

Same magnitudes as above, but negative numbers

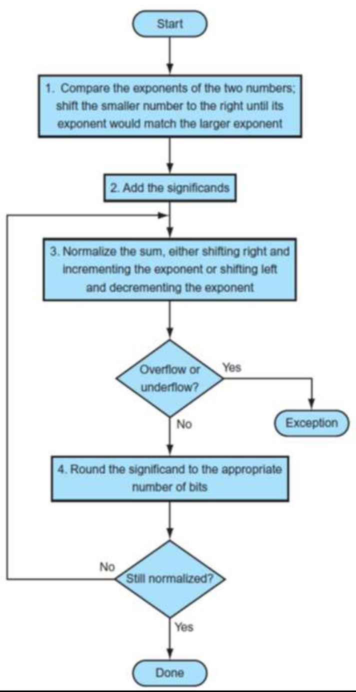
Floating Point Addition

- Consider a 4-digit binary example (0.5 + –0.4375)
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$

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) = 0.0625

Assumptions of associativity may fail

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00



Floating Point Multiplication

- 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
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 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 × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$