TE2003B
SoC Design: Computer organisation & architecture
Computer Arithmetic

Isaac Pérez Andrade

ITESM Guadalajara
School of Engineering & Science
Department of Computer Science
February – June 2021

References

The following material has been adopted and adapted from

Patterson, D. A., Hennessy, J. L., *Computer Organization and design: The hardware/software interface – ARM edition*, Morgan Kaufmann, 2017.

S. L. Harris and D. M. Harris, *Digital design and computer architecture - ARM edition*, Morgan Kaufmann, 2016.

J. Yiu, *The definitive guide to ARM Cortex-M0 and Cortex-M0+ processors*, Second edition, Elsevier, 2015.

Arithmetic for Computers

Operations on integers

- Addition and subtraction
- Multiplication and division
- Dealing with overflow

Floating-point real numbers

- Representation and operations

Integer operations

Two's complement review

Assume two's complement format

- Q: What's the range (minimum and maximum values that can be represented) of an N-bit two's complement number?

  A: $\left[-(2^{(N-1)}), 2^{(N-1)} - 1\right]$

  - For example, an 8-bit two's complement number may represent values in the range

  $[-2^{8-1}, 2^{8-1} - 1] = [-2^7, 2^7 - 1] = [-128,127]$

Overflow & underflow

- Q: What is **overflow**?

  A: A condition when the result of a calculation **exceeds** the **maximum** value that can be represented in a numeric format.

- Q: What is **underflow**?

  A: A condition when the result of a calculation is **smaller** than the **minimum** value that can be represented in a numeric format.

- Sometimes, the term overflow is used for describing both conditions.

Addition

Example: $7 + 6$

	(0)	(0)	(1)	(1)	(0)		(Carries)
. . .	0	0	0	1	1	1	(+7)
. . .	0	0	0	1	1	0	(+6)
. . .	(0) 0	(0) 0	(0) 1	(1) 1	(1) 0	(0) 1	(+13)

Overflow if result out of range

- Adding +ve and –ve operands, no overflow
- Adding two +ve operands

  Overflow if result sign is 1
- Adding two –ve operands

  Overflow if result sign is 0

Integer addition

- Example: Adding two 4-bit two's complement numbers

5 + 1

```
+5:   0101
+1:   0001
----------
+6:   0110
```

3 + 6

```
+3:   0011
+6:   0110
----------
-7:   1001
```

Overflow:+9 and -9 can not be represented in 4-bit two's complement.

−2 + 5

```
-2:   1110
+5:   0101
----------
+3:   0011
```

−7 +(−1)

```
-7:   1001
-1:   1111
----------
-8:   1000
```

−3 +(−6)

```
-3:   1101
-6:   1010
----------
+7:   0111
```

Subtraction

Integer subtraction

Addition with negation of second operand

Example: $7 - 6 = 7 + (-6)$

+7:	0000 0000 ... 0000 0111
–6:	1111 1111 ... 1111 1010
+1:	0000 0000 ... 0000 0001

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Integer subtraction

- Example: Subtracting two 4-bit two's complement numbers

5 – (+1)

```
+5:   0101
-1:   1111
----------
+4:   0100
```

–2 – (–5)

```
-2:   1110
+5:   0101
----------
+3:   0011
```

–3 – (+6)

```
-3:   1101
-6:   1010
----------
+7:   0111
```

7 –(–1)

```
+7:   0111
+1:   0001
----------
-8:   1000
```

Overflow: -9 and +8 can not be represented in 4-bit two's complement.

Addition & subtraction overflow summary

- Overflow conditions for additions and subtraction in two's complement.

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

Multiplication

- Start with long-multiplication approach

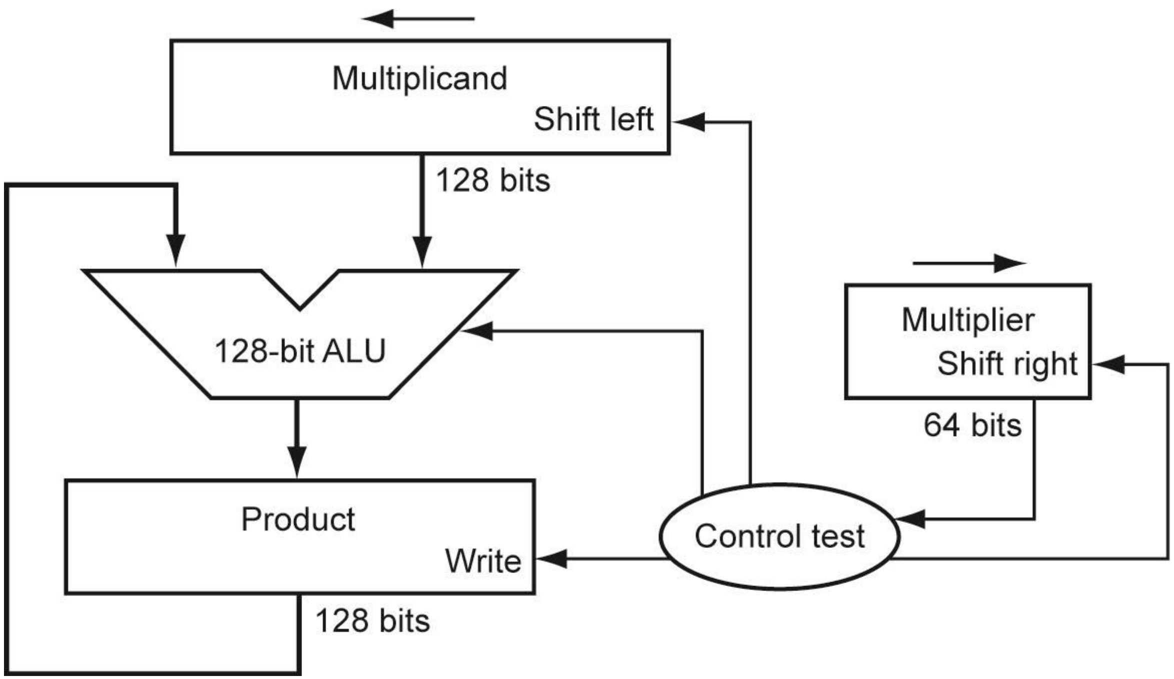
Assume we want to multiply two 64-bit numbers

multiplicand → 1000

multiplier → × 1001

```
     1000
  ×  1001
 --------
     1000
    0000
   0000
  1000
 --------
 01001000
```

product → 01001000



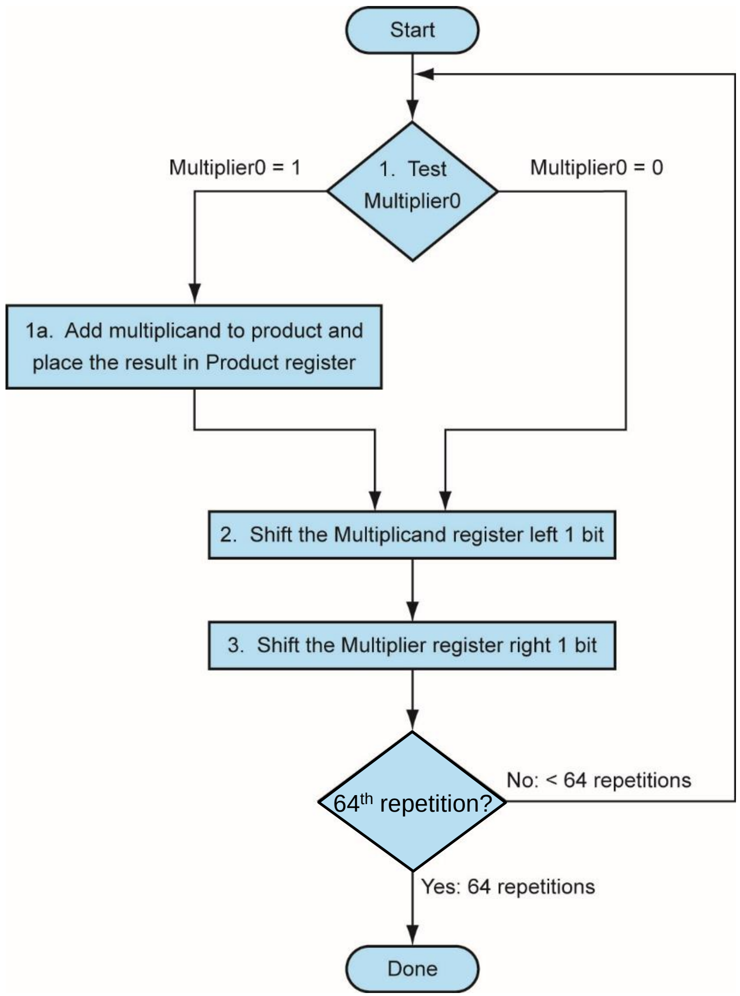
Length of product is the sum of operand lengths.

$$A_{M-bits} \times B_{N-bits} = X_{(M+N)-bits}$$

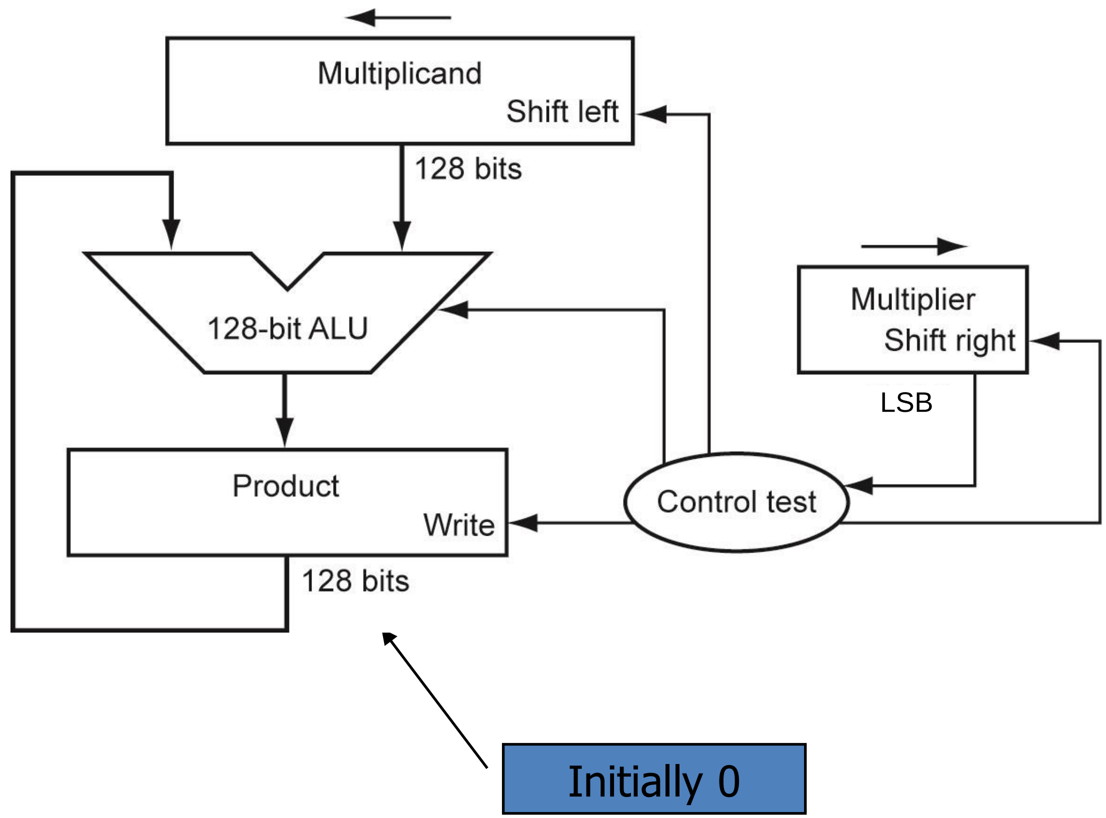
What can be improved in this architecture?

Multiplication hardware

There's one error in the flow chart. Can you spot it?

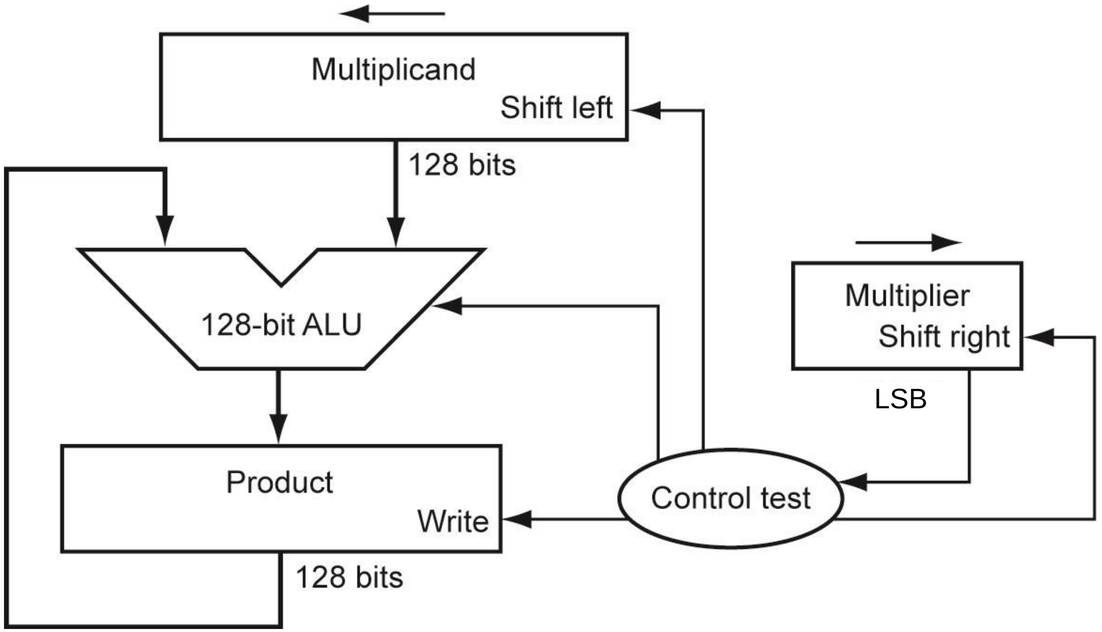


What can be improved in this architecture?



Multiplication hardware

This architecture has a major flaw.
Can you spot it?



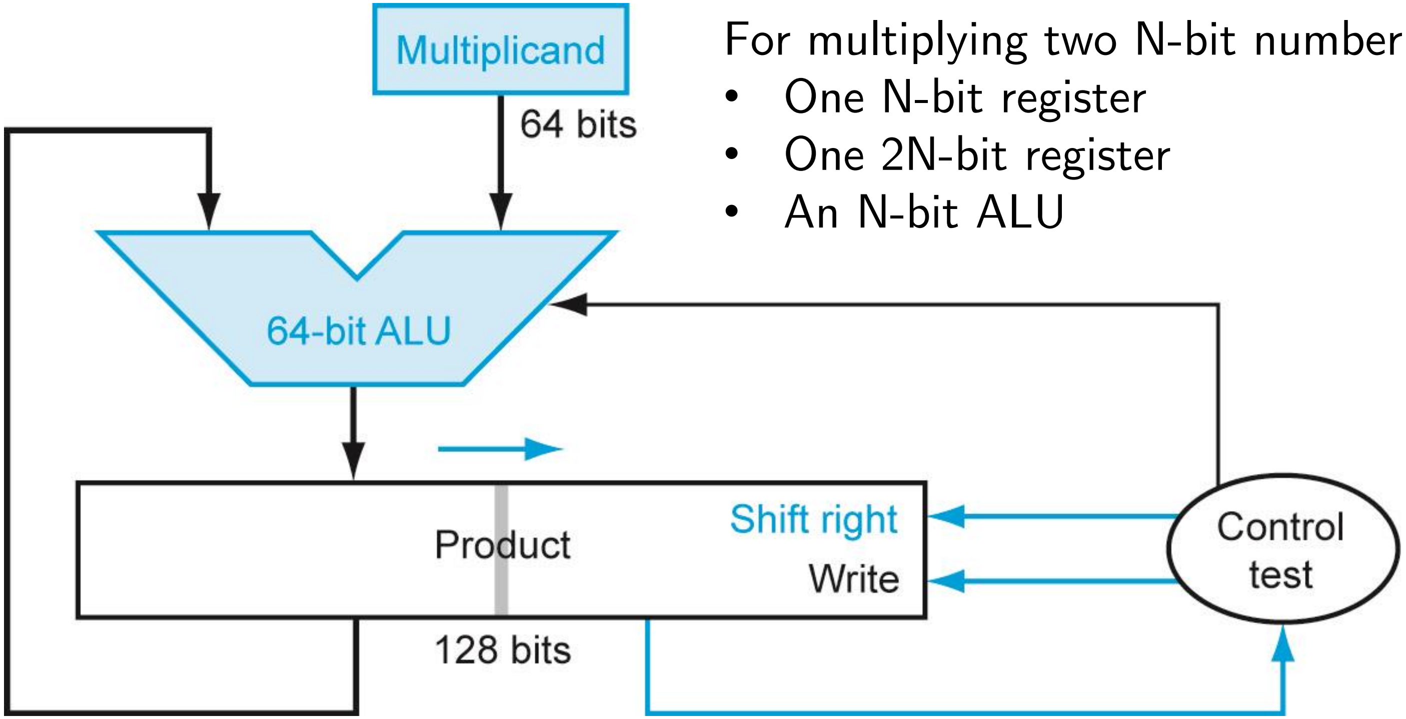
For multiplying two N-bit number, it requires:

- Two 2N-bit registers
- One N-bit register
- A 2N-bit ALU

This is a waste of resources!

Optimised multiplier

- Perform steps in parallel: add/shift

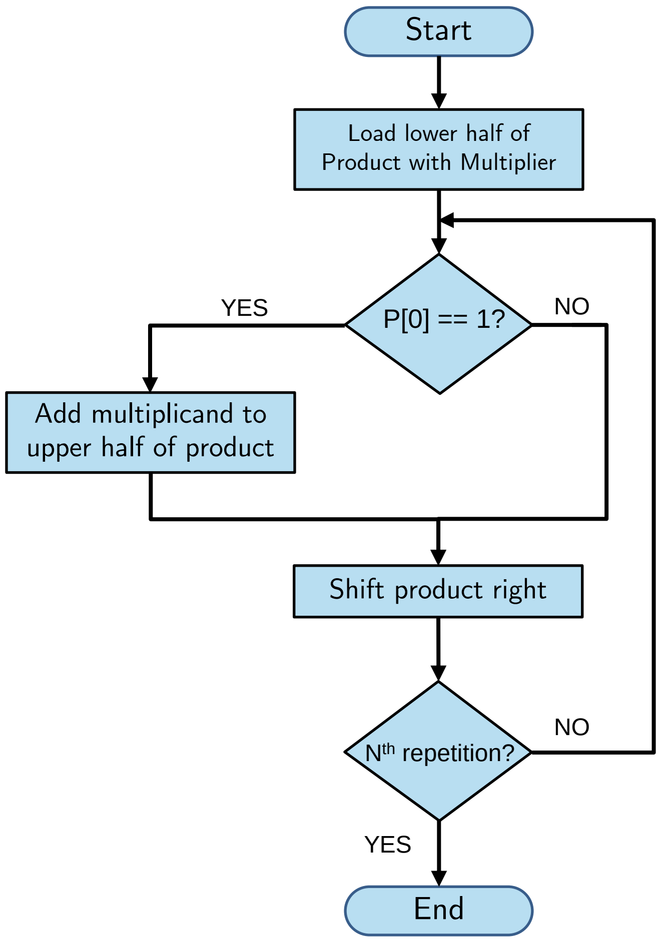


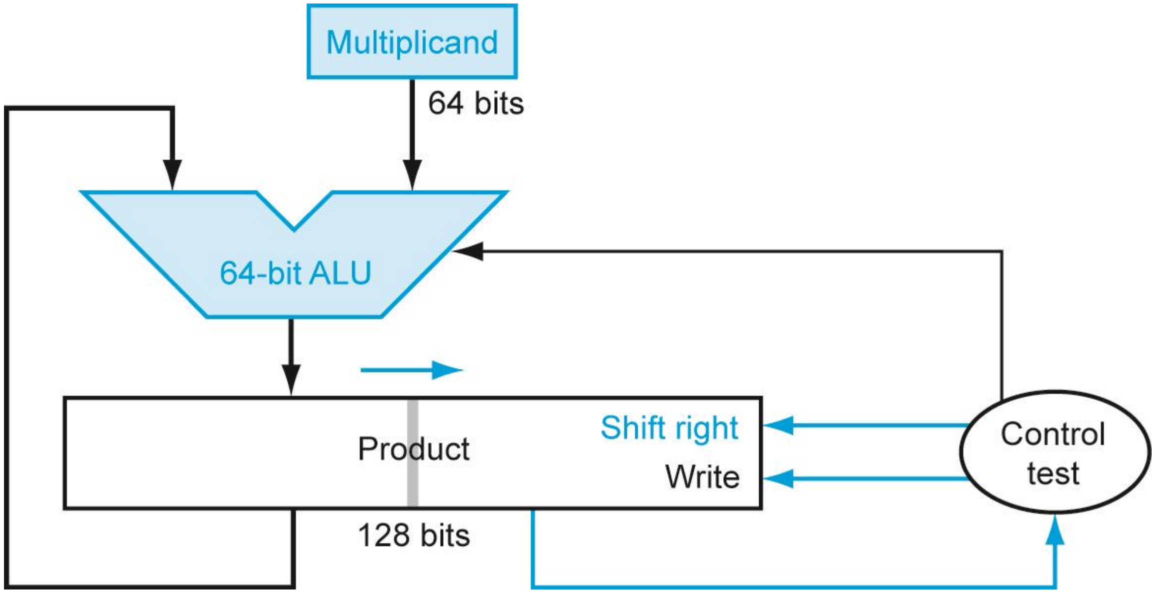
For multiplying two N-bit number, it requires:
- One N-bit register
- One 2N-bit register
- An N-bit ALU

One cycle per partial-product addition

- That's ok, if frequency of multiplications is low

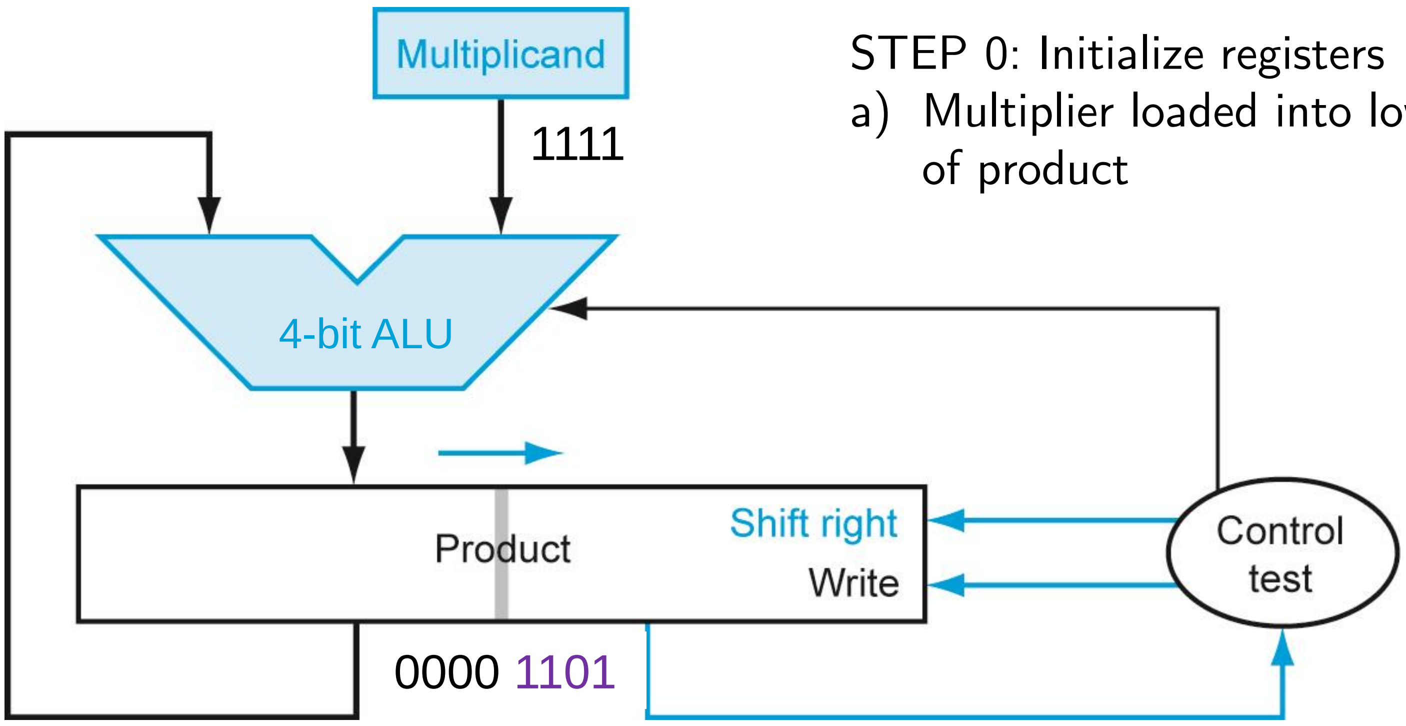
Optimised multiplier





Optimised multiplier - example

- Multiplicand $= 15_{10}$: 1111
- Multiplier $= 13_{10}$: 1101



STEP 0: Initialize registers

a) Multiplier loaded into lower half of product

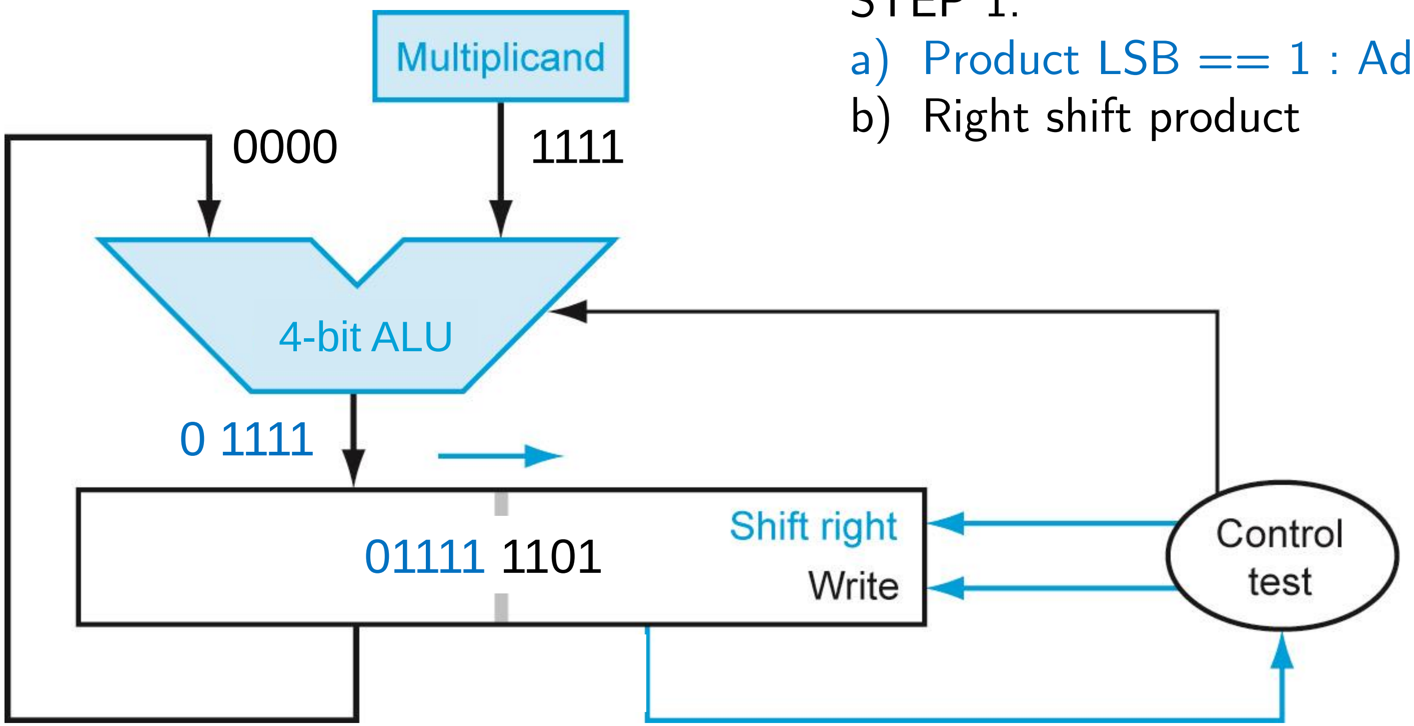
Optimised multiplier - example

- Multiplicand $= 15_{10}$: 1111
- Multiplier $= 13_{10}$: 1101

STEP 1:

a) Product LSB == 1 : Add

b) Right shift product



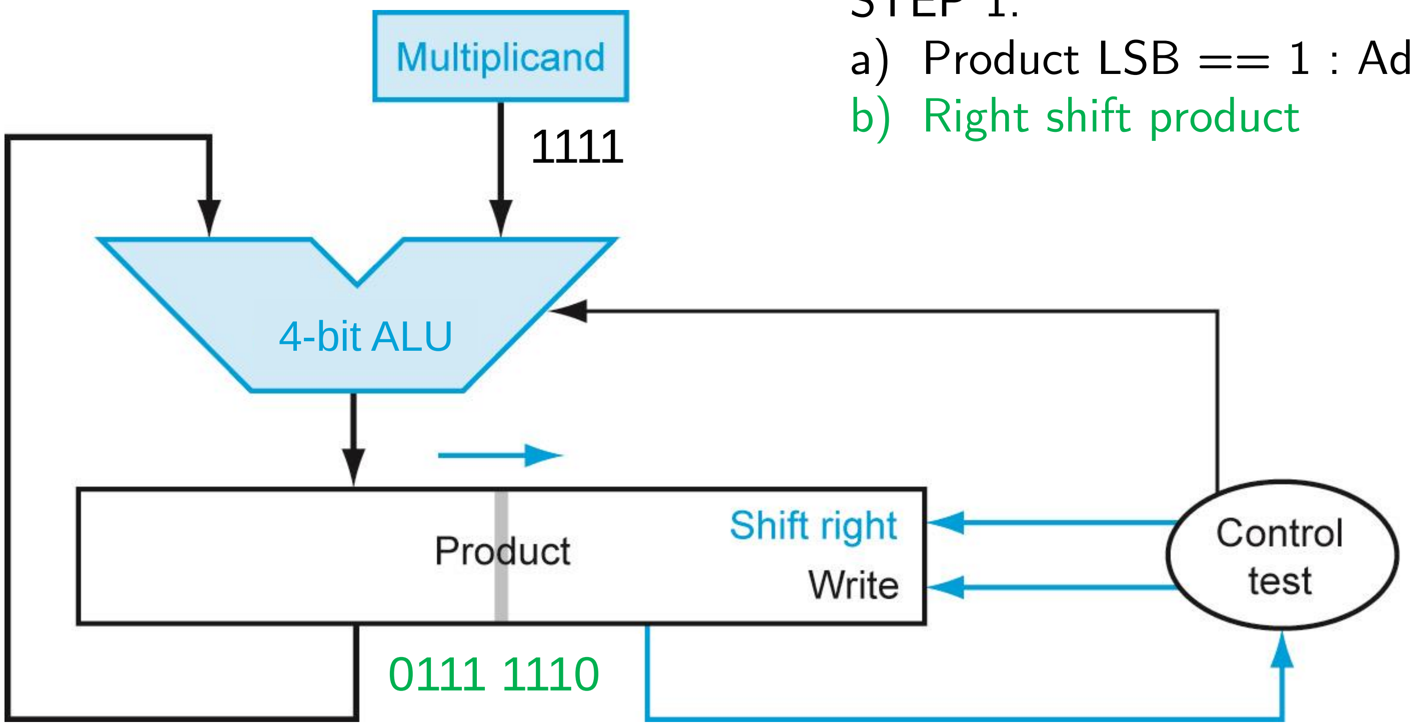
Optimised multiplier - example

- Multiplicand $= 15_{10}$: 1111
- Multiplier $= 13_{10}$: 1101

STEP 1:

a) Product LSB == 1 : Add

b) Right shift product



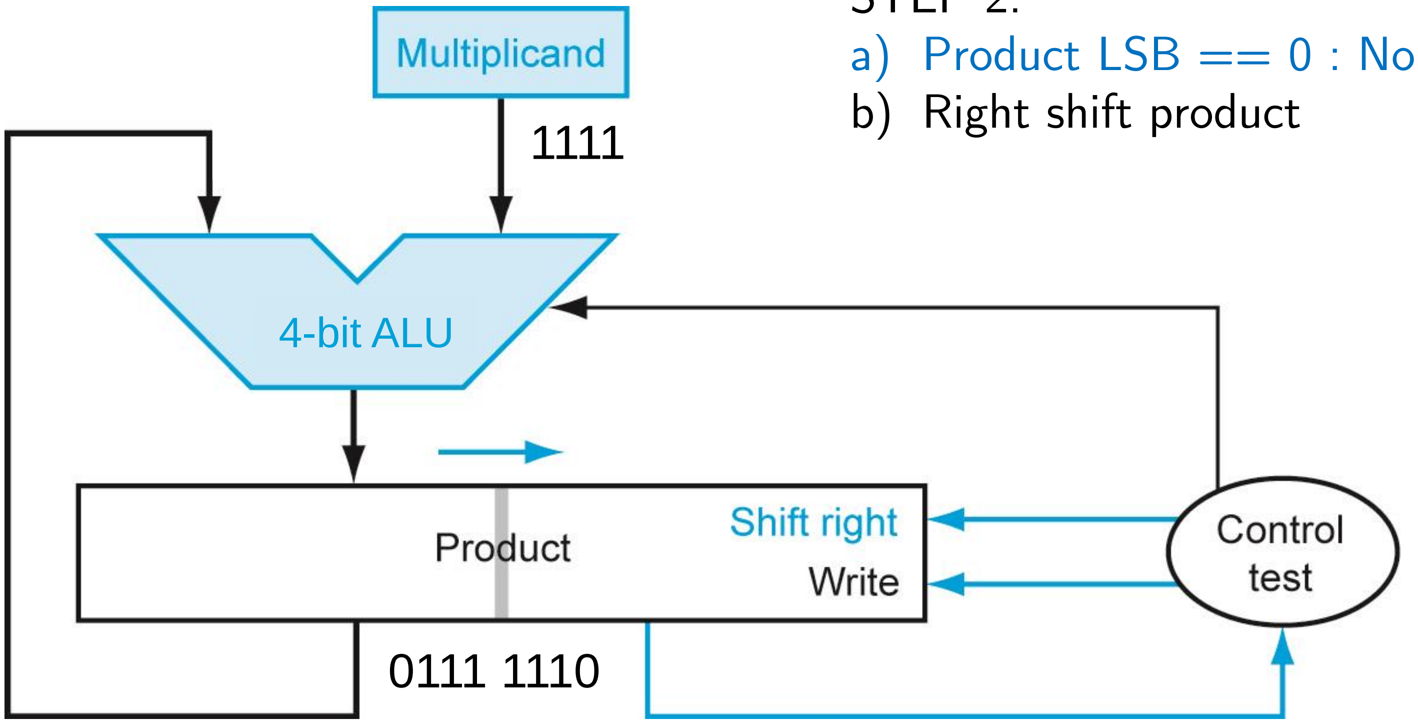
Optimised multiplier - example

- Multiplicand = 15_{10}: 1111
- Multiplier = 13_{10}: 1101

STEP 2:

a) Product LSB == 0 : No Add

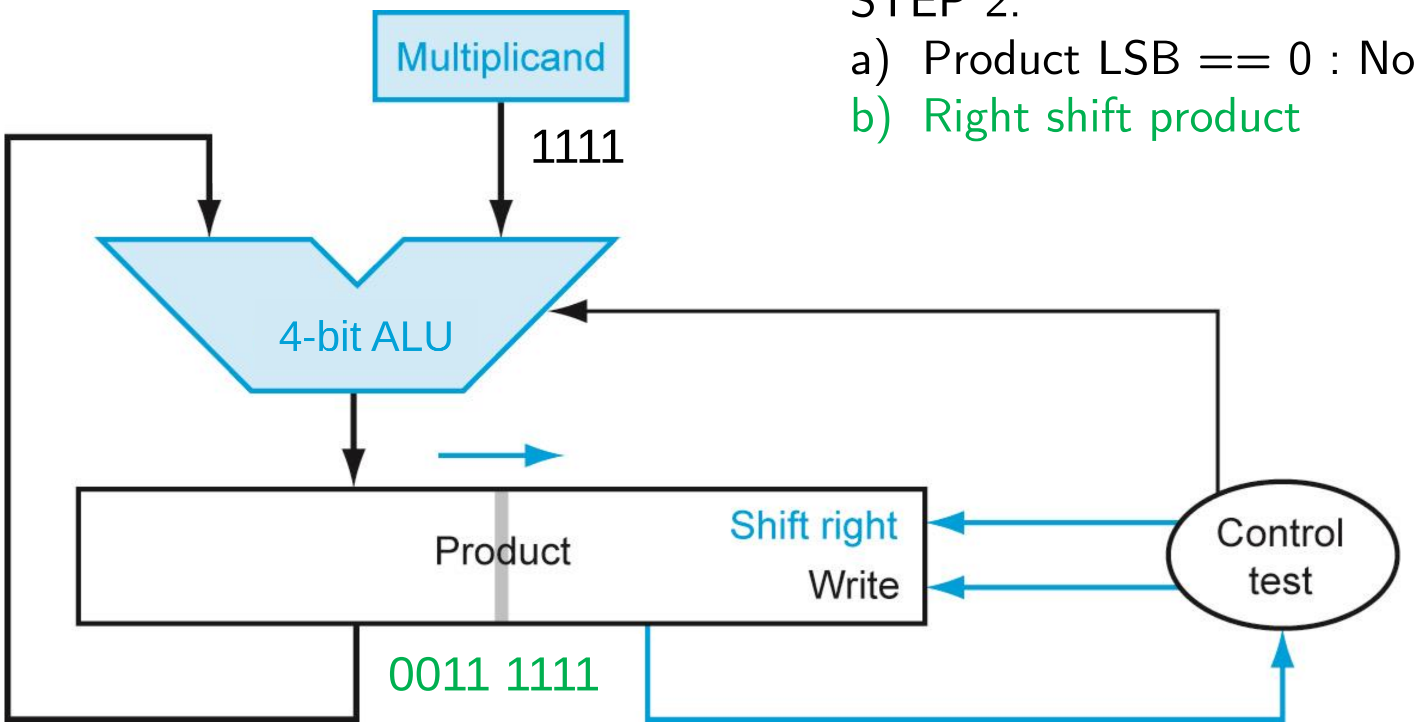
b) Right shift product



Optimised multiplier - example

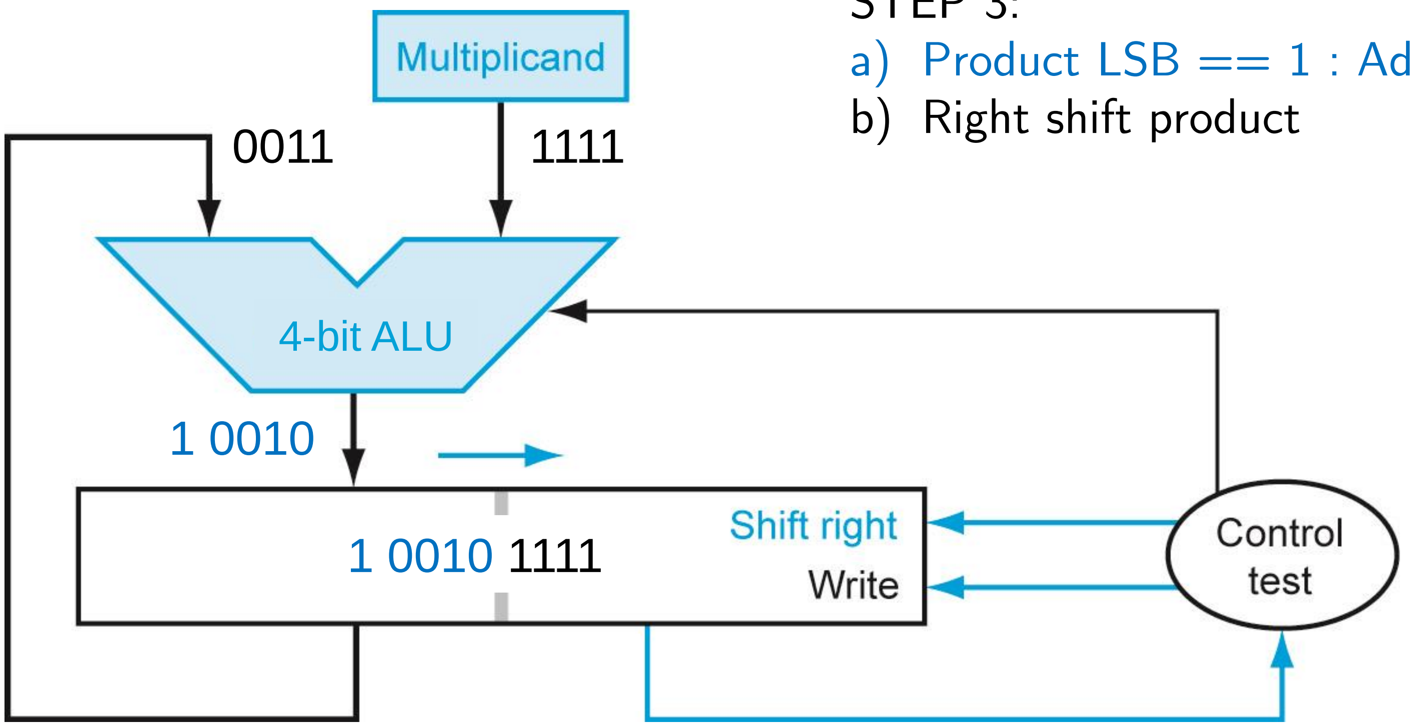
- Multiplicand $= 15_{10}$: 1111
- Multiplier $= 13_{10}$: 1101

STEP 2:
a) Product LSB == 0 : No Add
b) Right shift product



Optimised multiplier - example

- Multiplicand $= 15_{10}$: 1111
- Multiplier $= 13_{10}$: 1101



STEP 3:

a) Product LSB == 1 : Add

b) Right shift product

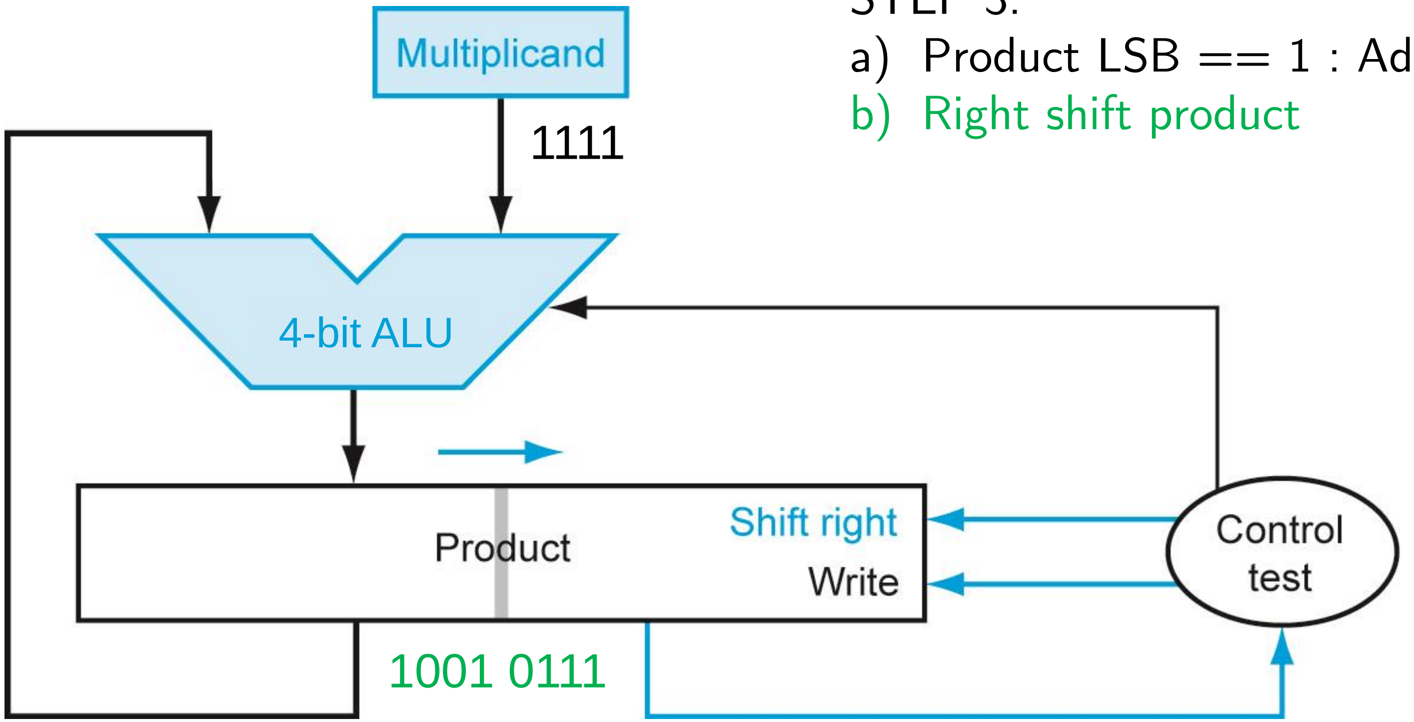
Optimised multiplier - example

- Multiplicand $= \mathbf{15_{10}}$: 1111
- Multiplier $= \mathbf{13_{10}}$: 1101

STEP 3:

a) Product LSB == 1 : Add

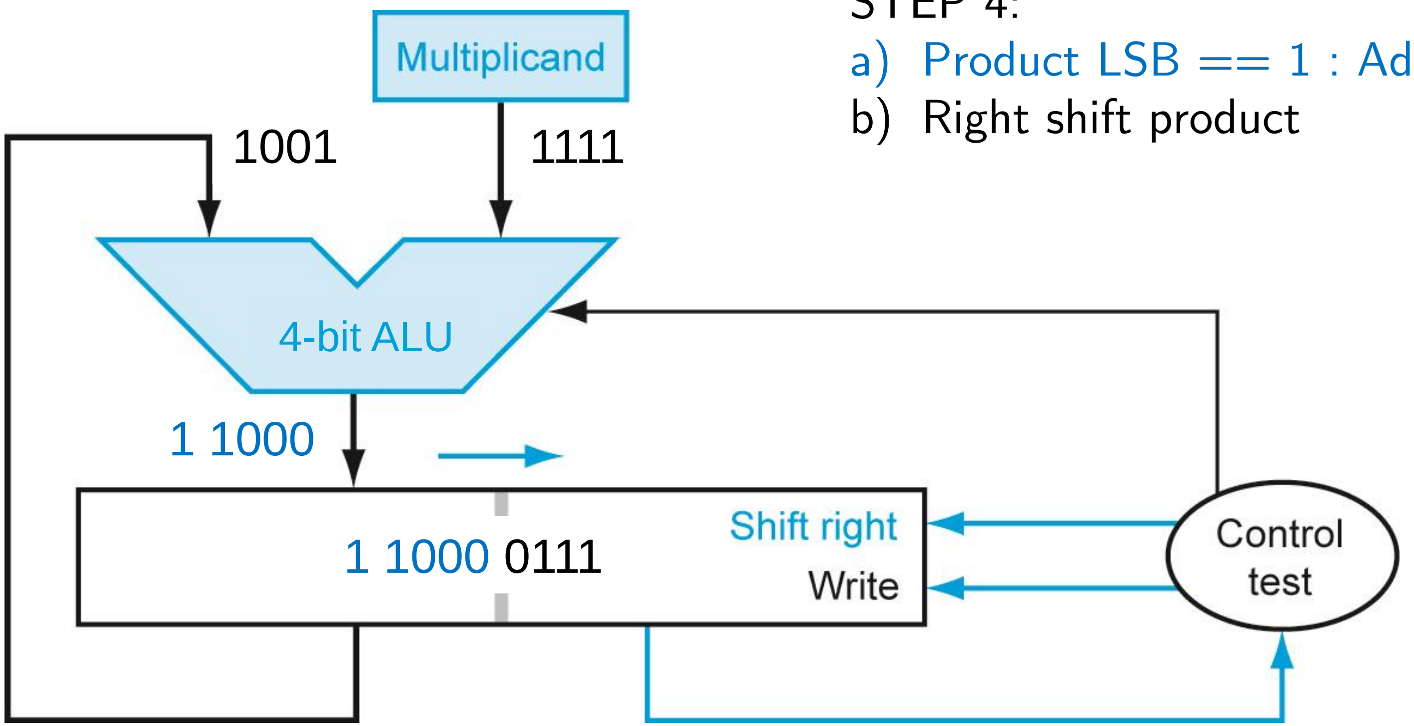
b) Right shift product



Optimised multiplier - example

- Multiplicand $= 15_{10}$: 1111
- Multiplier $= 13_{10}$: 1101

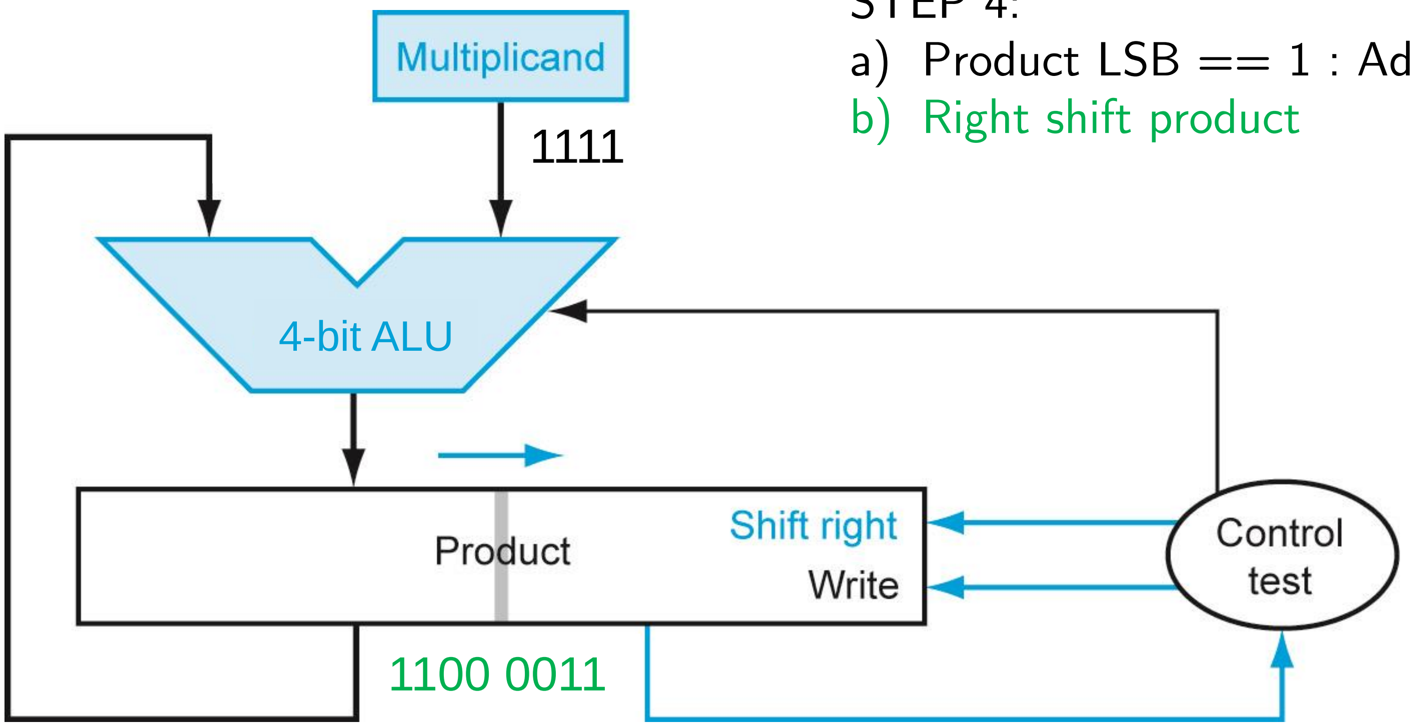
STEP 4:
a) Product LSB == 1 : Add
b) Right shift product



Optimised multiplier - example

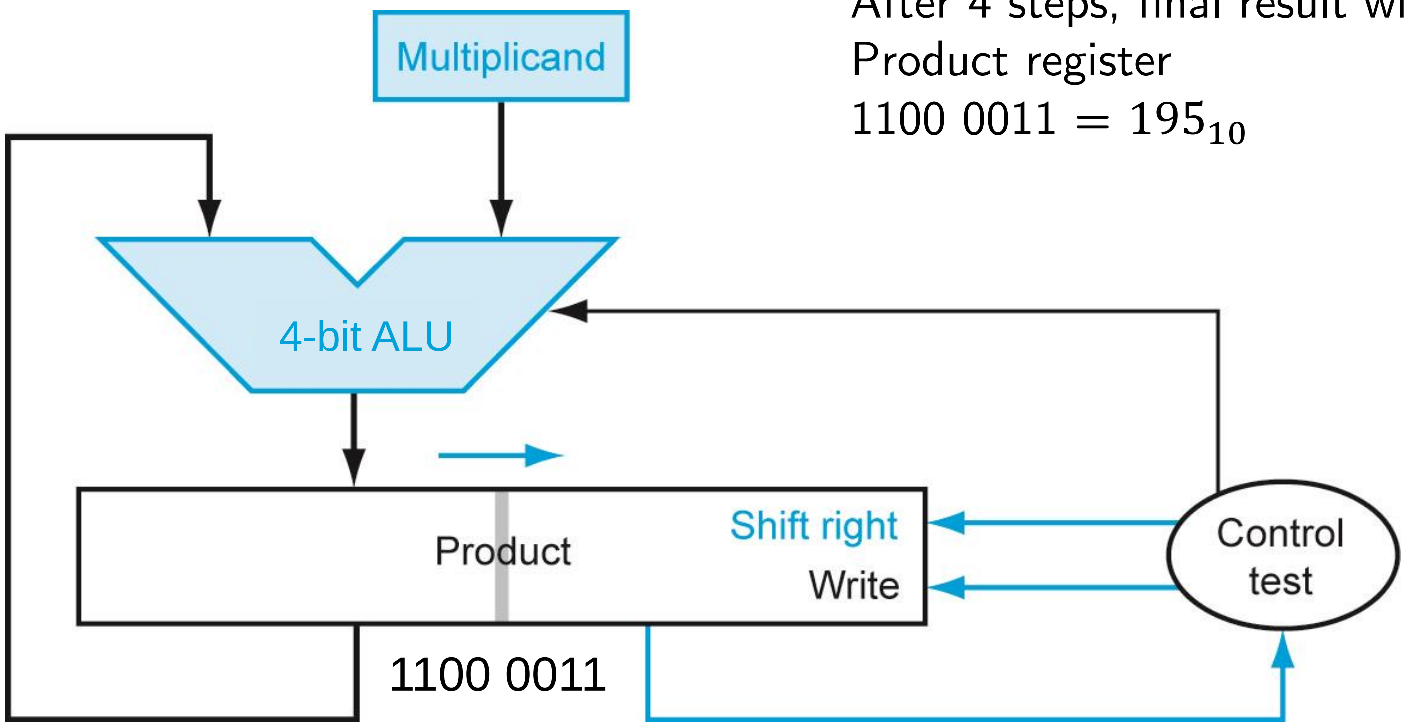
- Multiplicand $= \mathbf{15_{10}}$: 1111
- Multiplier $= \mathbf{13_{10}}$: 1101

STEP 4:
a) Product LSB == 1 : Add
b) Right shift product



Optimised multiplier - example

- Multiplicand $= \mathbf{15_{10}}$: 1111
- Multiplier $= \mathbf{13_{10}}$: 1101



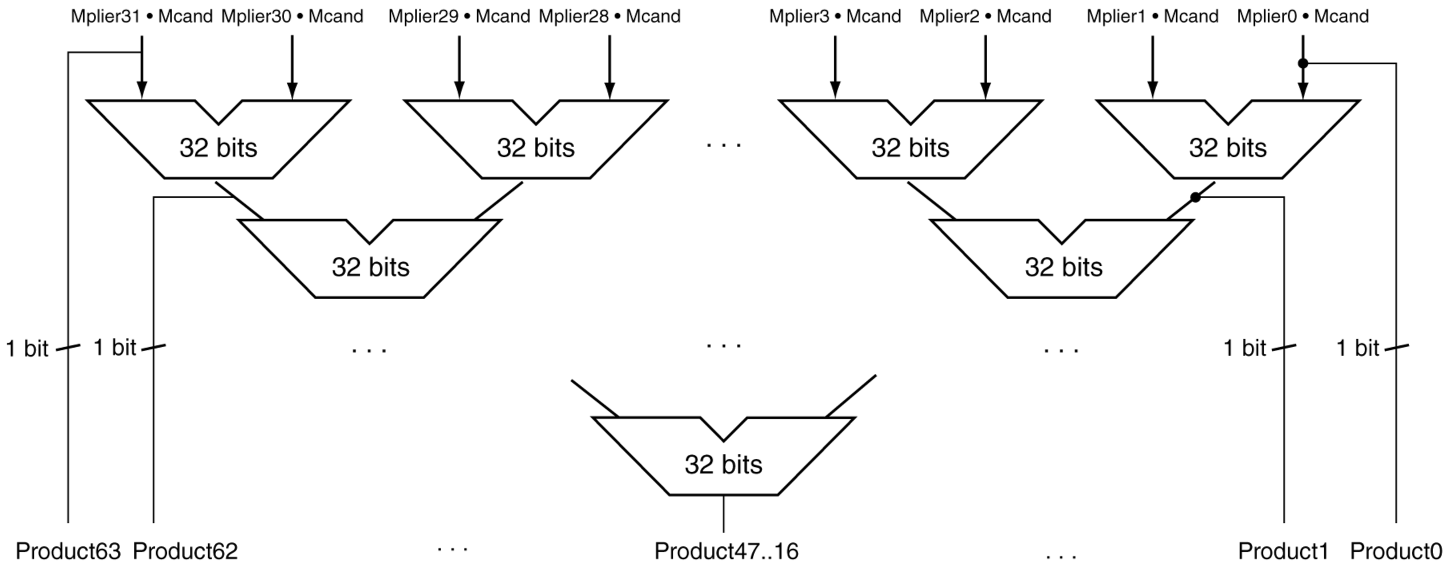
After 4 steps, final result will be in Product register
$1100\ 0011 = 195_{10}$

Signed multiplication

- So far, we've only dealt with unsinged operands.
- What happens in signed multiplication?
- For adding two signed N-bit numbers:
    1. Convert both multiplicand and multiplier to positive numbers and keep track of their respective sign.
    2. Apply multiplication algorithm N-1 times.
    3. Negate product if signs are not the same.
- Alternatively, previous algorithm works for signed numbers as long as shifts are performed using sign extension.

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



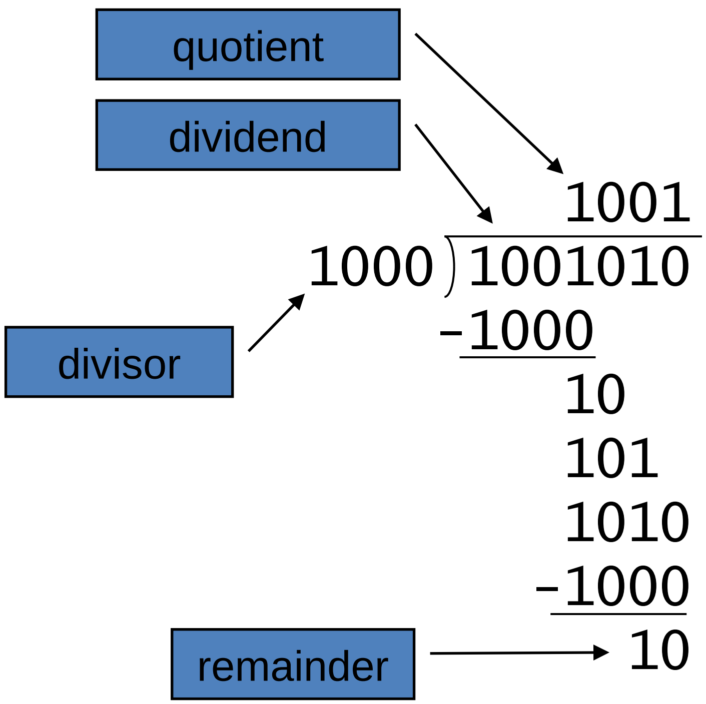
Use (N-1) N-bit adders in parallel, instead of a single N-bit adder (N-1) times.

Can be pipelined

- Several multiplication performed in parallel

Division

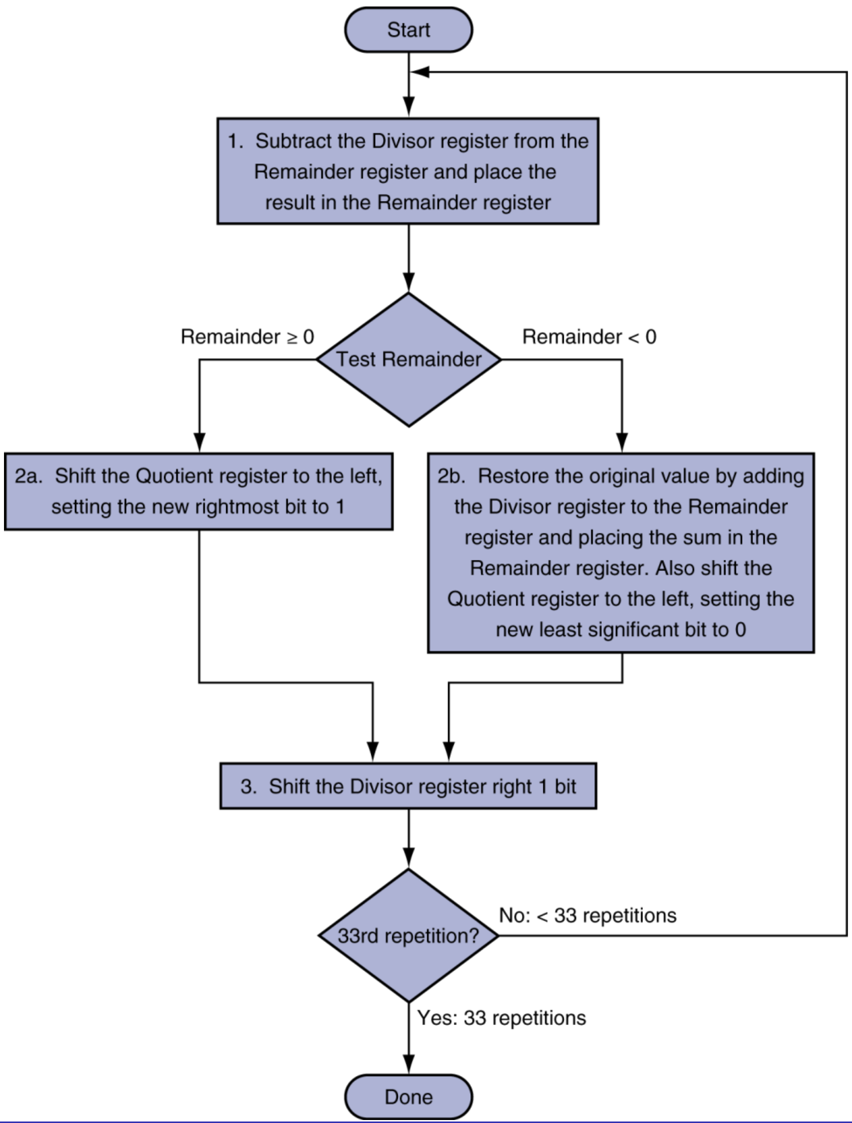
Division

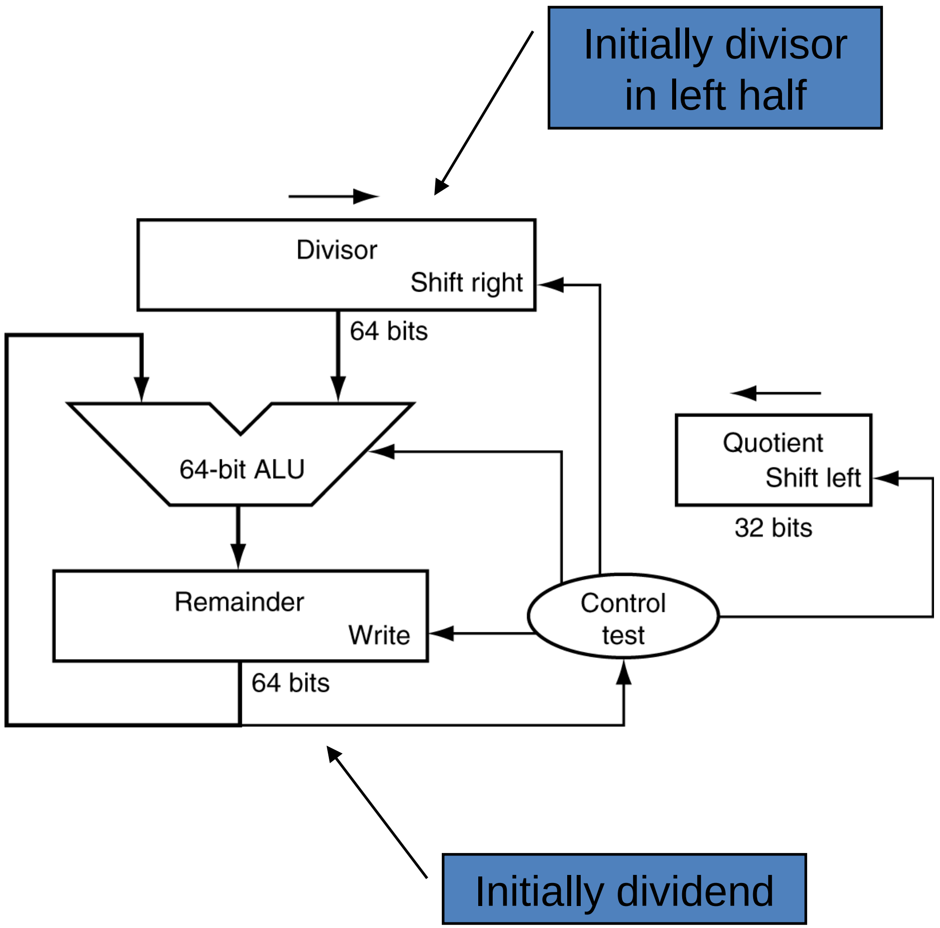


n-bit operands yield *n*-bit quotient and remainder

- Check for 0 divisor
- Long division approach
  - If divisor $\leq$ 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

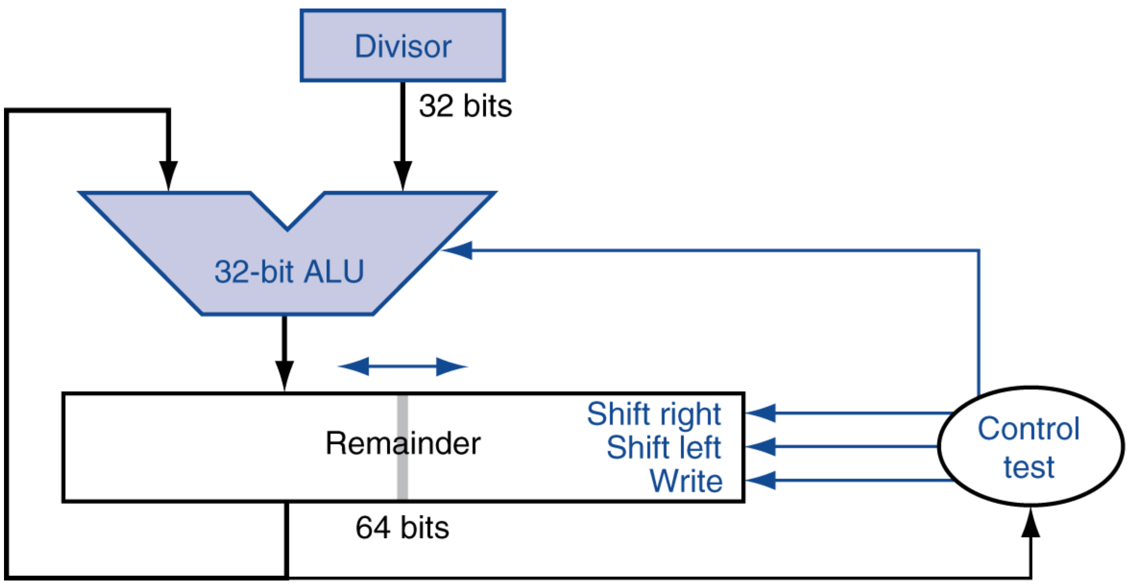
Division hardware





Optimized divider

- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both



Faster Division

- Can't use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers generate multiple quotient bits per step
  - Still require multiple steps

Fixed-point representation

Fixed-point introduction

- Real (fractional) numbers may not be represented with integer numbers.

```
integer a,b;
a = 1.5;
b = a + a; // b = ?
```

- Fixed-point representation allows real number representation with limited precision.
  - Qm.n representation
    - m → number of bits for representing integer part.
    - n → number of bits for fractional part.
    - Range $[-(2^{m-1}), 2^{m-1} - 2^{-n}]$
    - Resolution is 2^{-n}

Fixed-point

- Fixed-point representation is suitable for embedded applications requiring limited degree of fractional precision.
  - DOOM (1993 videogame) originally used a Q16.16 format for all non-integer operations https://doomwiki.org/wiki/Fixed_point

- What about high-precision applications?

Fixed-point limitations

- Example:
  - Consider Avogadro's number: 6.022×10^{23}
    - How many bits would you need to represent Avogadro's number?

$$\lceil \log_2(6.022 \times 10^{23}) \rceil = 79$$

  - What about a very small number such as Planck's constant: $6.62607004 \times 10^{-34}$ J · s
    - How many bits (fractional fixed-point) would you need to represent Planck's constant?

$$|\lceil \log_2(6.62607004 \times 10^{-34}) \rceil| = 110$$

  - We would need at least $79 + 110 = 189$ bits for representing both numbers.
    - Not feasible, waste of resources.
    - What if need even smaller or larger numbers?

Floating-point representation

Floating-point

- Scientific notation
  - A single digit to the left of the decimal point
  - $+1.12345 \times 10^{-7}$
  - -123.456×10^{9}
    - **Sign**
    - **Mantissa/significant**
    - **Exponent**

normalized

not normalized

- Normalized scientific notation
  - Absolute value of integer part m is in the range

$$[1, 10) \rightarrow 1 \leq m \leq 9$$

- Binary numbers may also be represented in scientific notation

$$1.0_2 \times 2^{-1} = 0.1_2$$

0.5_{10}

$$0.1_2 \times 2^{0} = 0.1_2$$

normalized

not normalized

Floating-point

- As the name suggest, binary point is not fixed.
- Representation for non-integral numbers
  - Including very small and very large numbers
  - $(-1)^{\text{sign}} 1.\text{mantissa} \times 2^{(\text{exponent}-\text{bias})}$
  - For simplicity, we'll show the exponent in decimal.
- Programming languages refer to this representation as `float` and `double` types.

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)
- Simplifies exchange of data, arithmetic and increases accuracy.

IEEE Floating-point format

1 bit	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
sign	exponent	mantissa

Implicit 1

$$x = (-1)^{\text{sign}}\, 1.\text{mantissa} \times 2^{(\text{exponent} - \text{bias})}$$

- sign: ($0 \Rightarrow$ non-negative, $1 \Rightarrow$ negative)
- Significand: 1.mantissa
- Normalized significand: $1.0 \le |\text{significand}| < 2.0$
- exponent $=$ actual exponent $+$ bias
  - Ensures exponent is unsigned
  - Single: bias $= 127$
  - Double: bias $= 1203$

IEEE Floating-point format

1 bit	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
sign	exponent	mantissa

$$x = (-1)^{\text{sign}} 1.\text{mantissa} \times 2^{(\text{exponent}-\text{bias})}$$

- First example: 0.5 to single-precision floating-point

$$+0.5_{10} = +0.1_2 = +1.0_2 \times 2^{-1}$$

For simplification, we are using decimal notation for the exponent

- From this, we can gather the following information:
  - Sign: 0
  - Actual exponent: -1
  - Mantissa: 0 (normalized)
- Everything together:
  - Adjusted exponent$= -1 + 127 = 126_{10}$ or 01111110_2
  - Normalized floating-point:

$$(-1)^0 1.0 \times 2^{(126-127)}$$

0 01111110 00000000000000000000000

IEEE Floating-point format

1 bit	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
sign	exponent	mantissa

$$x = (-1)^{\text{sign}} 1.\text{mantissa} \times 2^{(\text{exponent}-\text{bias})}$$

- First example: 0.5 to single-precision floating-point

$$+0.5_{10} = (-1)^{0} 1.0 \times 2^{(126-127)}$$

This 1 is not actually required here

0 01111110 00000000000000000000000

- What about 0.5 in double-precision?

$$0.5_{10} = (-1)^{0} 1.0 \times 2^{(1022-1023)}$$

0 01111111110 0000000000000000000000000000000000000000000000
000000

Floating-point example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - sign = 1
  - mantissa = $1000\ldots00_2$
  - exponent = -1 + bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-point example

1 bit	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
sign	exponent	mantissa

- What number is represented by the single-precision float

  11000000101000000000000000000000

  - sign = 1
  - mantissa = $01000...00_2$
  - exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1.01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Remember that this 1 is always implied!

Single precision range

Exponents 00000000 and 11111111 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
- $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double precision range

Exponents 00000000000 and 11111111111 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
- $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-point precision

- Relative precision
  - Single: approx. $2^{-23} \rightarrow \approx 7$ decimal digits
  - Double: approx. $2^{-52} \rightarrow \approx 16$ decimal digits

Floating-point special representation

- Denormal numbers
  - In normalised numbers, significand have an implicit leading 1
    $$x = (-1)^{\text{sign}} 1.\text{mantissa} \times 2^{(\text{exponent}-\text{bias})}$$
  - Denormal numbers have a leading 0 in the significand.
  - Biased exponent is 0.
  - These numbers allow to represent numbers smaller than the smaller normalised number, as well as special representation such as $\pm\infty$ and NaN $(\mathbf{0} \div \mathbf{0})$.

Denormal numbers

$$x = (-1)^{\text{sign}} \times (0.\text{mantissa}) \times 2^{0-\text{bias}}$$

- Smaller than normal numbers.

- Zero
  - sign $= 0,1$
  - biased exponent $= 0$
  - mantissa $= 0$

$$x = (-1)^{\text{sign}} \times (0 + 0) \times 2^{-\text{bias}} = \pm 0.0$$

Two representations of 0.0!

Denormalized numbers

Smallest denormalized value

- Single precision: $\pm 2^{-23} \times 2^{-126} \approx 1.4 \times 10^{-45}$
- Double precision: $\pm 2^{-52} \times 2^{-1022} \approx 4.9 \times 10^{-324}$

Largest denormalized value

- Single precision: $\pm(1 - 2^{-23}) \times 2^{-126} \approx \pm 1.17 \times 10^{-38}$
- Double precision: $\pm(1 - 2^{-52}) \times 2^{-1022} \approx \pm 2.22 \times 10^{-308}$

Infinities and NaNs

- Exponent = 111…1, mantissa = 000…0
  - $\pm\infty$
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111…1, mantissa $\neq$ 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 special formats summary

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

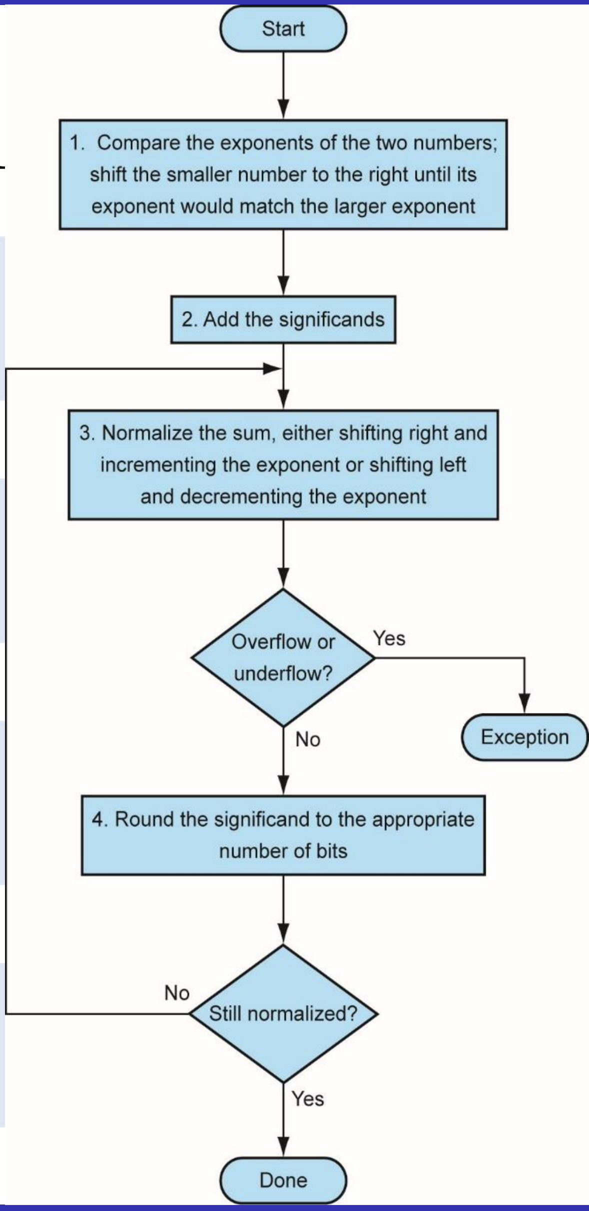
Floating-point adition

Floating-point addition

- Consider a 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}

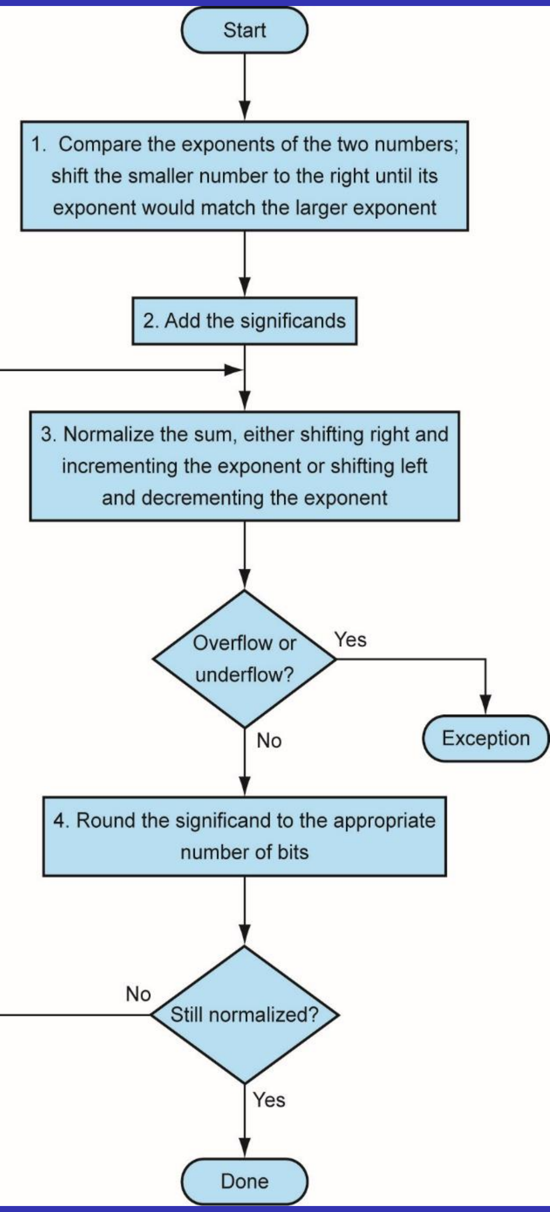


Floating-point addition

- Consider a floating-point example

  $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ or (0.5+-0.4375)

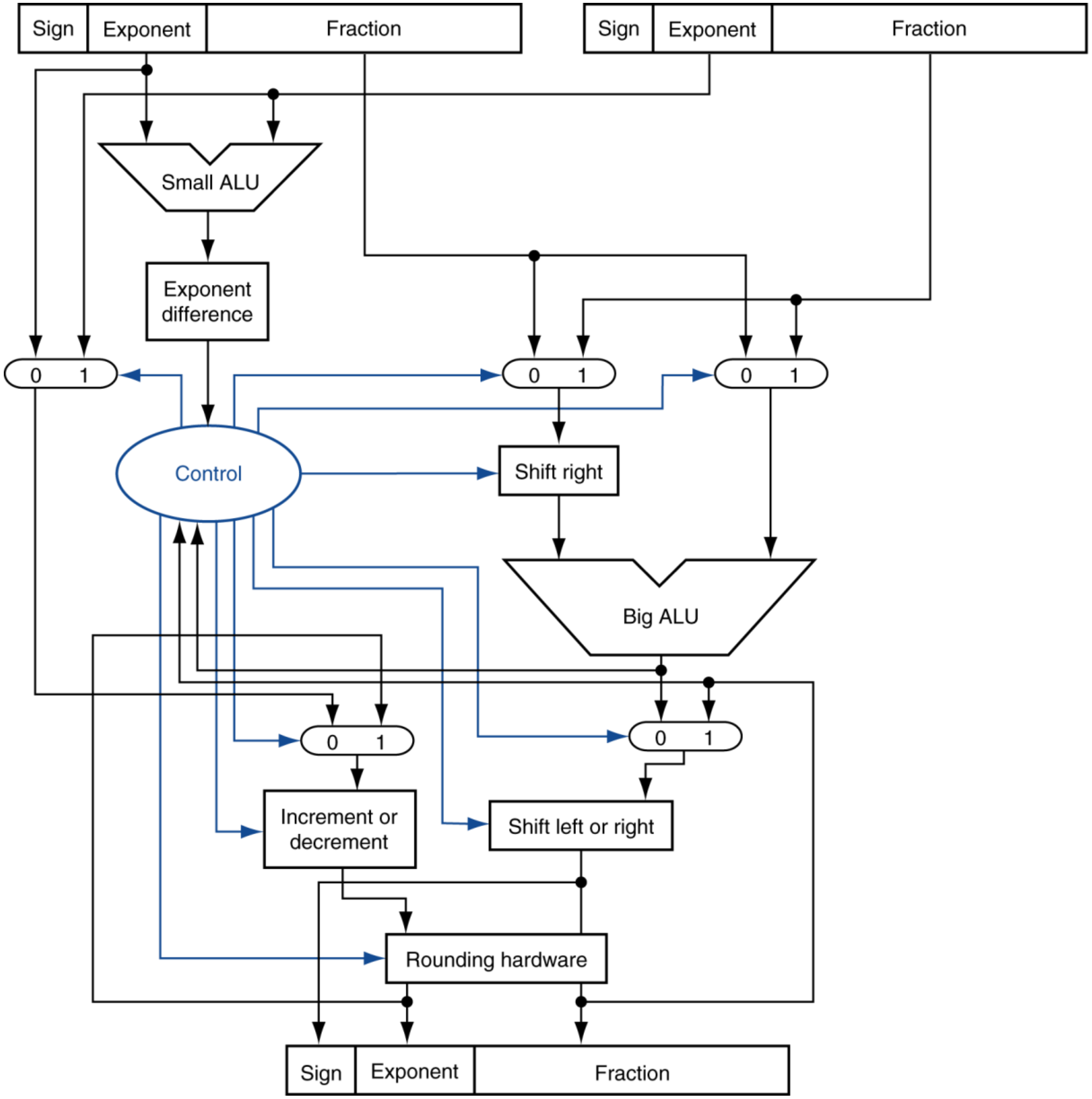
1. Align decimal points	Shift number with smaller exponent $1.000_2 \times 2^{-1} + -0.1110_2 \times 2^{-1}$
2. Add significands	$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
3. Normalize result & check for over/underflow	$1.0_2 \times 2^{-4}$
4. Round and renormalize if necessary	$1.0_2 \times 2^{-4} = 0.0625$

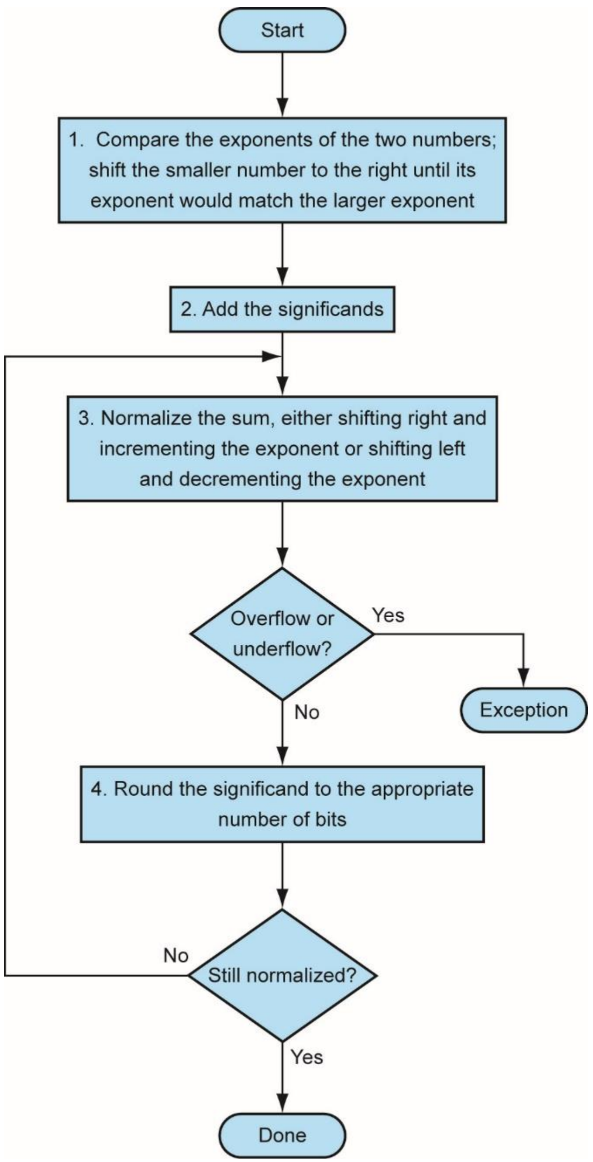


Floating-point adder hardware

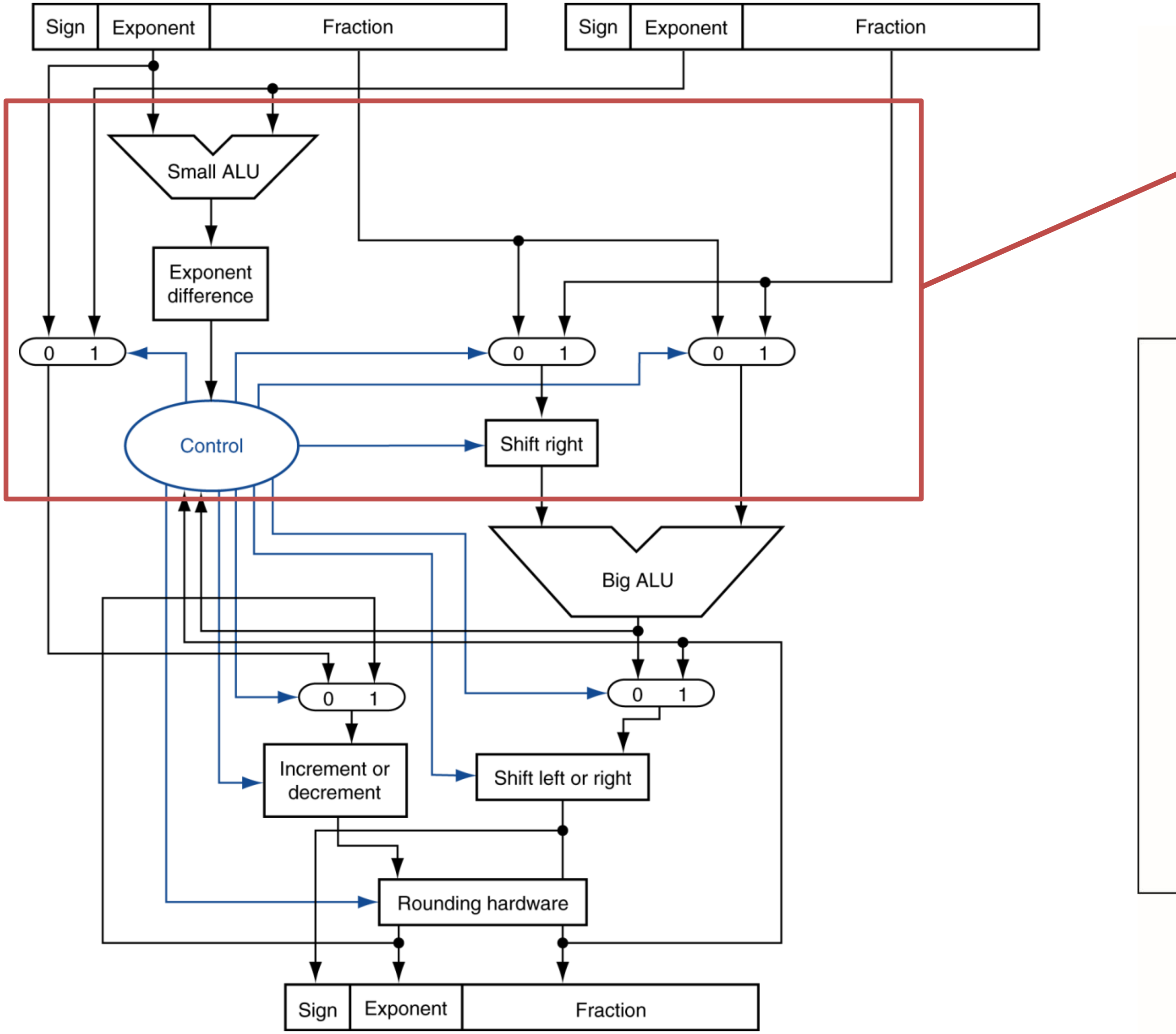
- Much more complex than integer adder.
- Doing it in one clock cycle would take too long.
  - Much longer than integer operations.
  - Slower clock would penalize all instructions.
- Floating-point adder usually takes several cycles
  - Can be pipelined

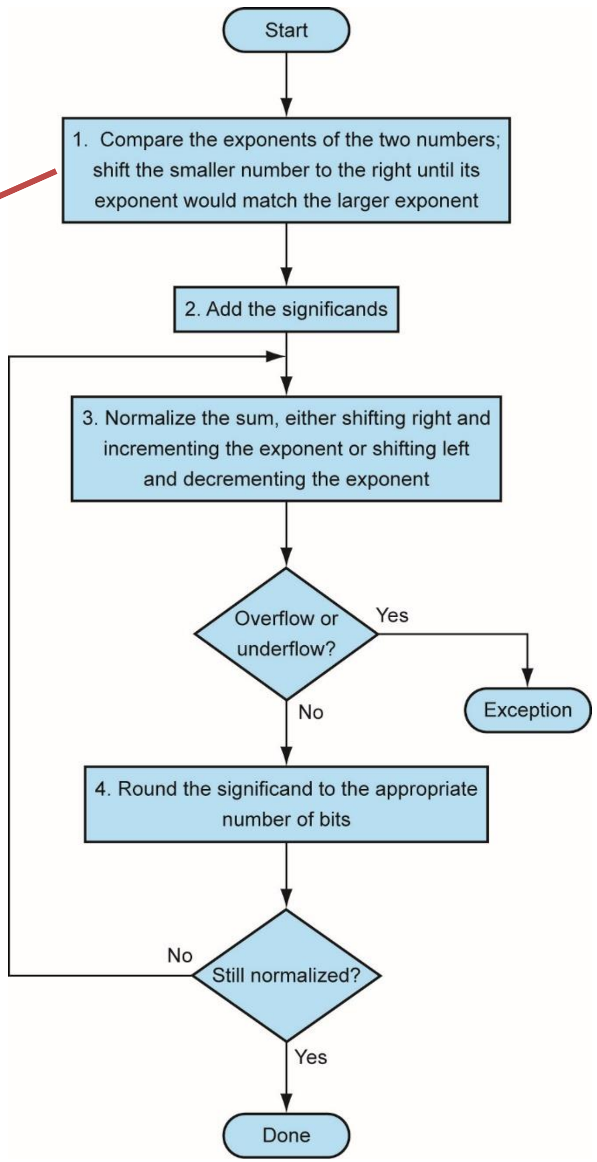
Floating-point adder hardware



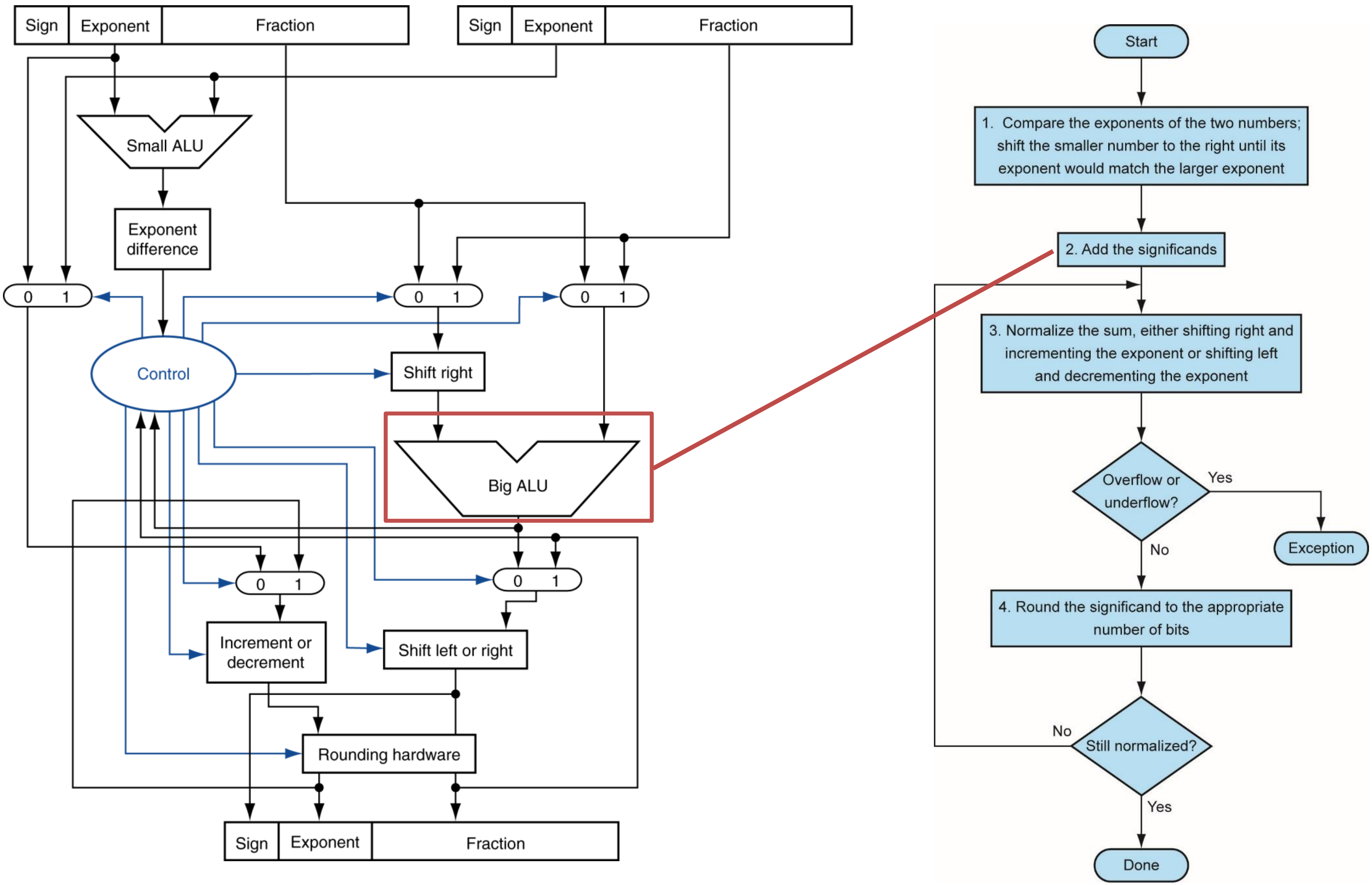


Floating-point adder hardware

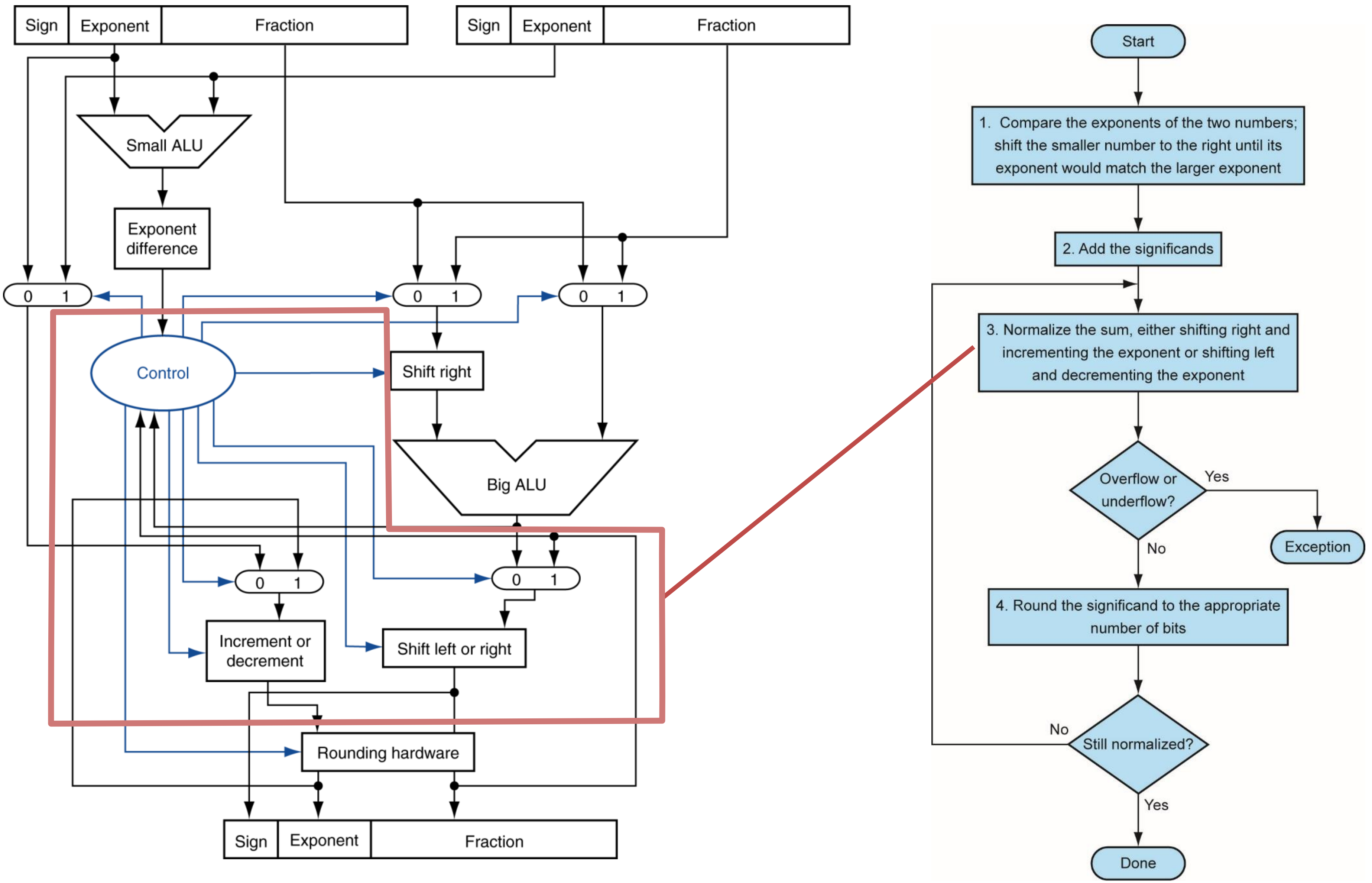




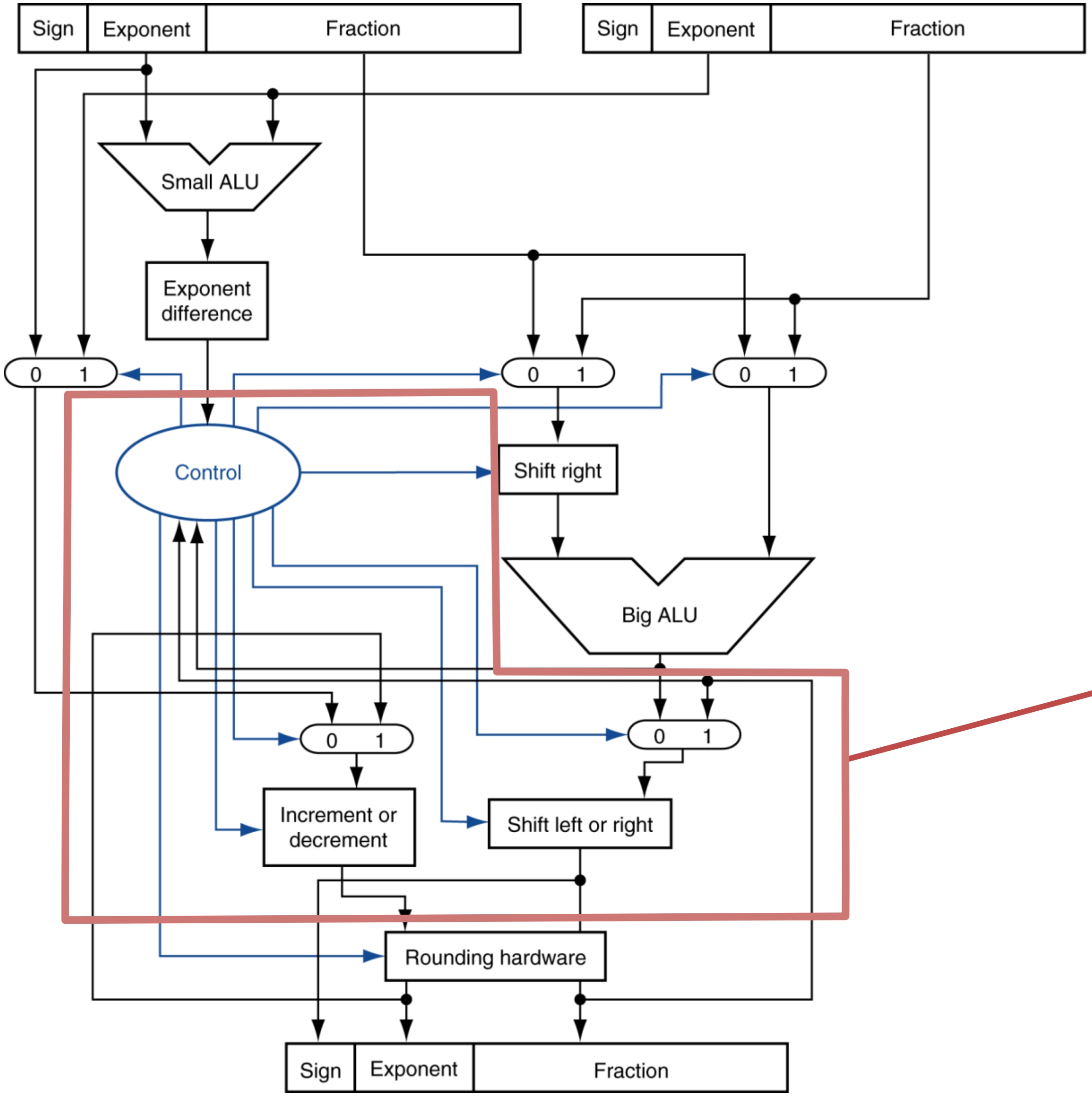
Floating-point adder hardware

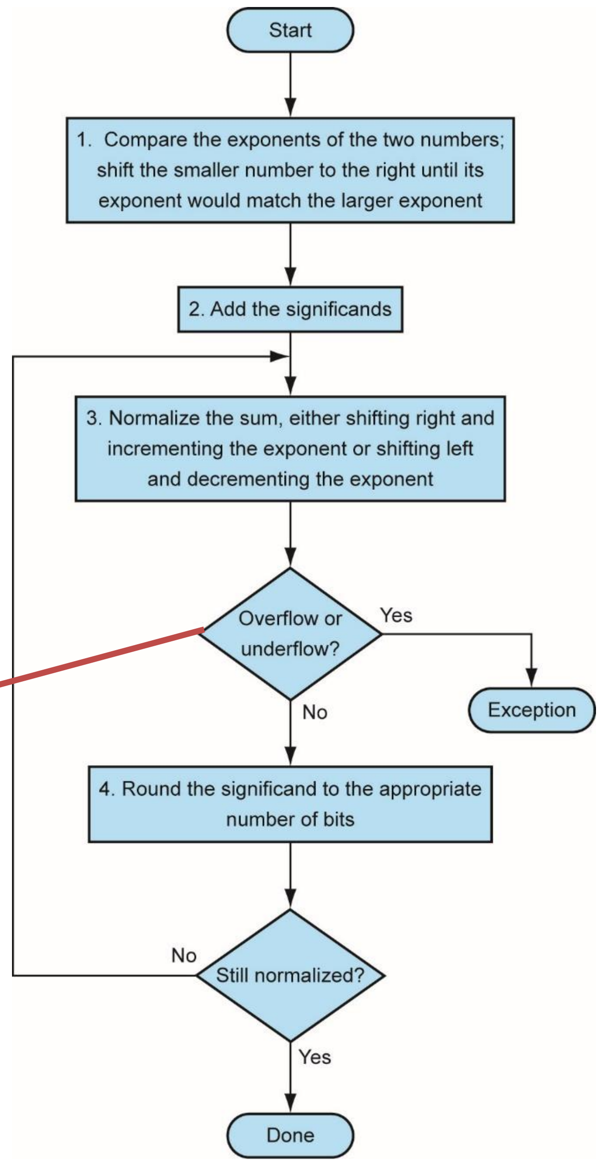


Floating-point adder hardware

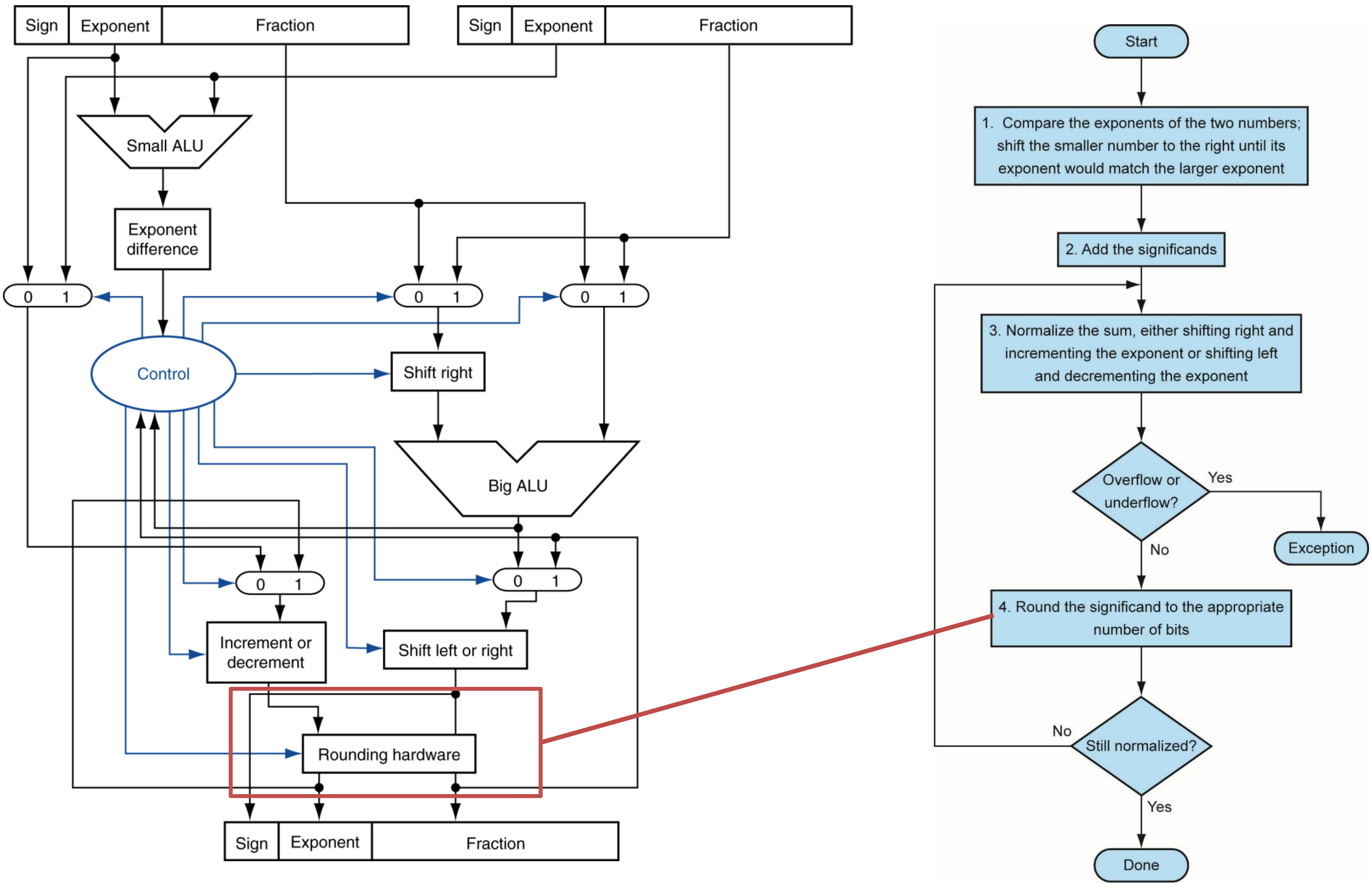


Floating-point adder hardware

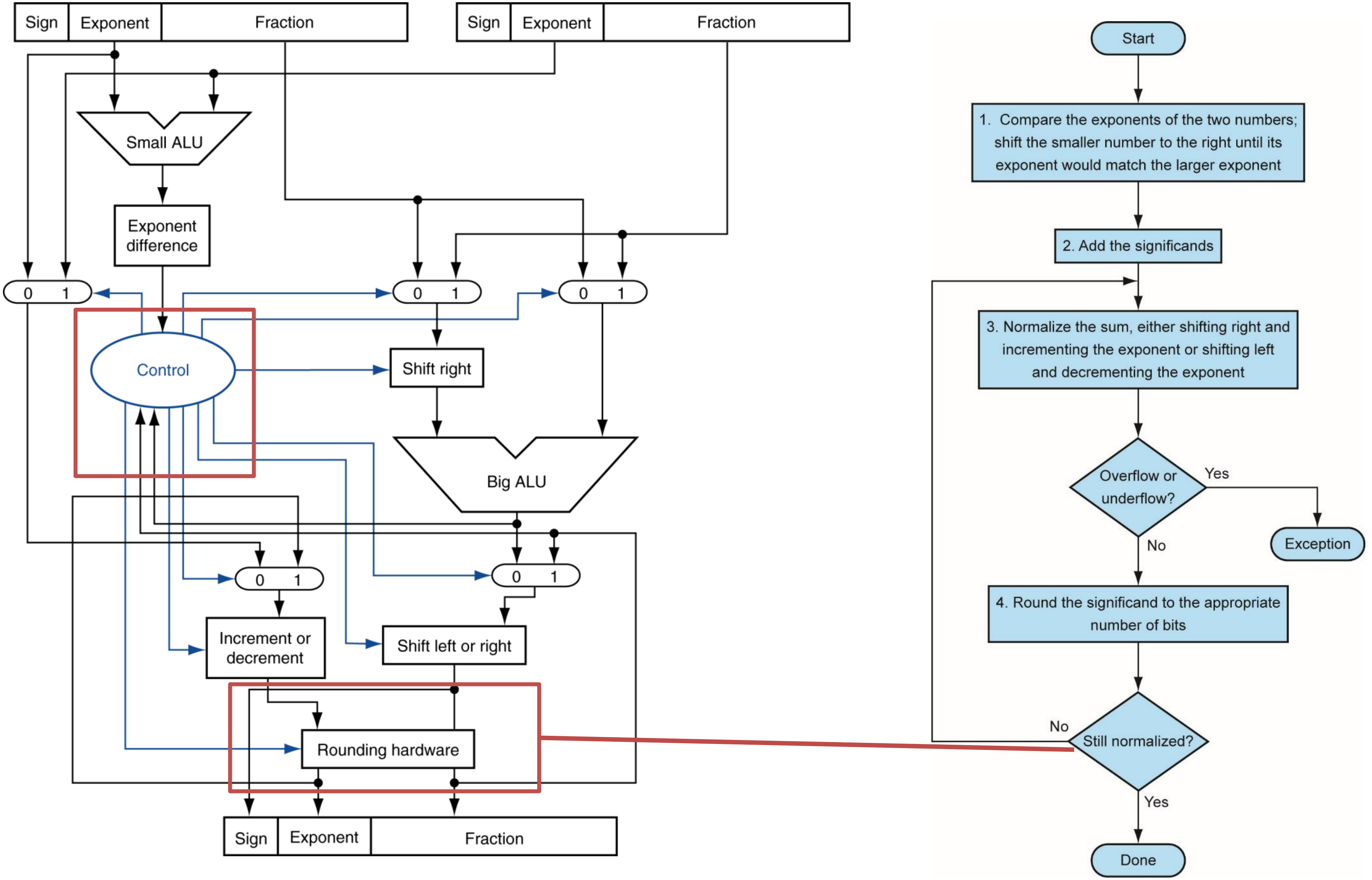




Floating-point adder hardware



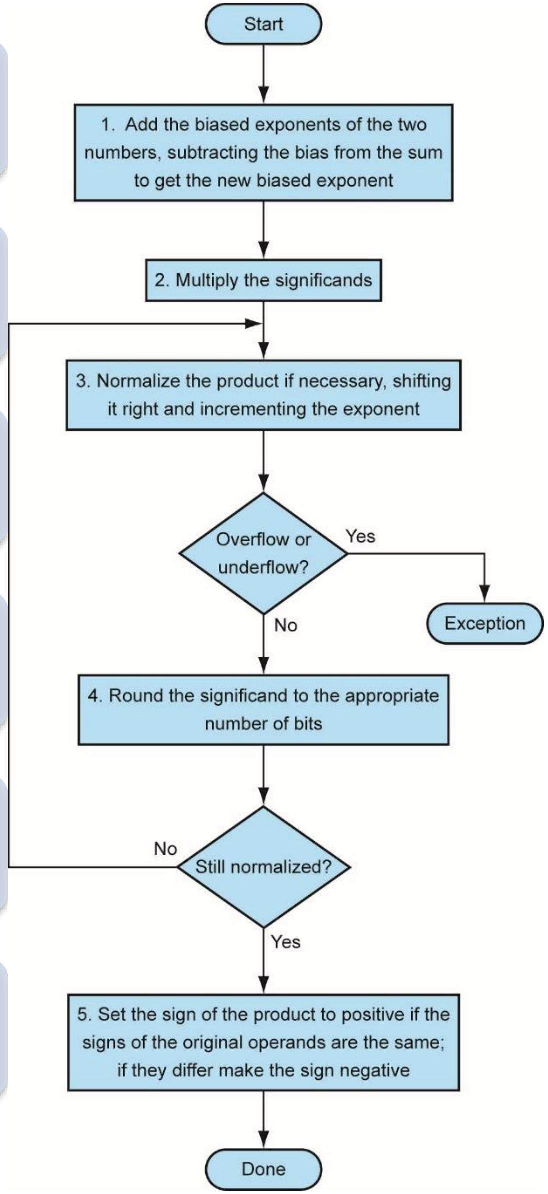
Floating-point adder hardware



Floating-point multiplication

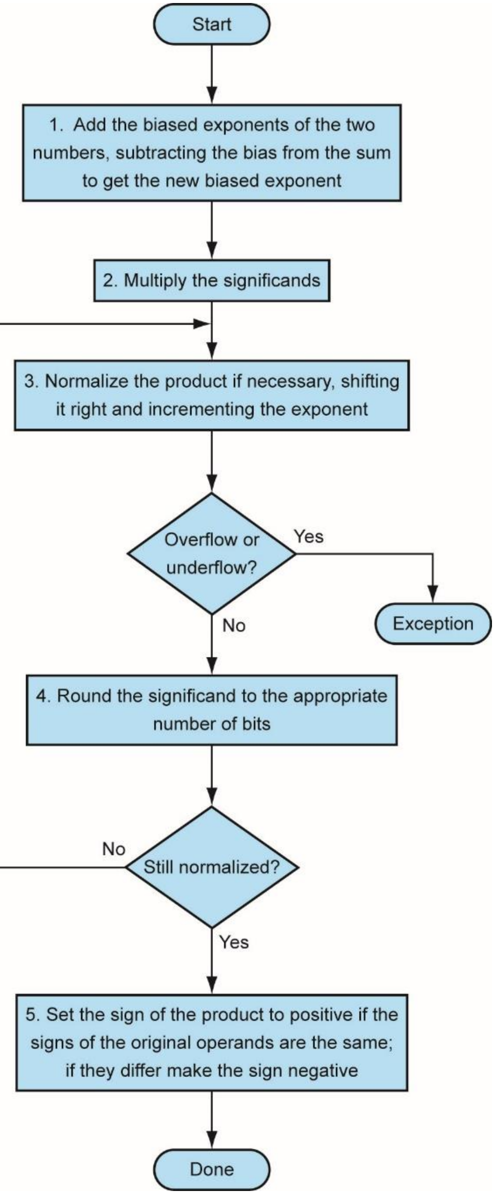
Floating-point multiplication

Consider a 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	1.110×9.200 $= 10.212 \Rightarrow 10.212 \times 10^{5}$
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	$+1.021 \times 10^{6}$



Floating-point multiplication

Consider a floating-point example	-14.25×3.125 in decimal, or $-1.11001_2 \times 2^3 \times 1.1001_2 \times 2^1$
1. Add exponents	• Unbiased: $3 + 1 = 4$ • Biased: $(3 + 127) + (1 + 127) - 127$ $= 4 + 254 - 127 = 4 + 127$
2. Multiply significands	$-1.11001_2 \times 1.1001_2$ $= 10.110010001_2$ $\Rightarrow 10.110010001_2 \times 2^4$
3. Normalize result & check for over/underflow	$1.0110010001_2 \times 2^5$ with no over/underflow
4. Round and renormalize if necessary	$1.0110010001_2 \times 2^5$(no change)
5. Determine sign of result from signs of operands	$-1.0110010001_2 \times 2^5 = -44.53125$



Floating-point multiplication

- What's the floating-point of the previous result?

$$-14.25_{10} \times 3.125_{10} = -44.53125_{10}$$

$$(-1.11001_2 \times 2^3) \times (1.1001_2 \times 2^1) = -1.0110010001 \times 2^5$$

  - sign $= 1$
  - exponent
    - Single precision: $5 + 127 = 132 \rightarrow 10000100$
    - Double precision: $5 + 1023 = 1028 \rightarrow 10000000100$
  - mantissa $= 01100100010000000000000$
  - Floating point representation:

Single precision: 1 10000100 01100100010000000000000

Double precision: 1 10000000100 0110010001000 ... 00

Floating-point arithmetic hardware

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

Floating-point rounding

Floating-point rounding

- Let's try to represent 0.1_{10} in floating-point.
  1. Represent 0.1_{10} in binary
     1. Integer part is 0
     2. Fractional part is:

$$
\begin{aligned}
0.1 \times 2 &= 0 + 0.2 \\
0.2 \times 2 &= 0 + 0.4 \\
0.4 \times 2 &= 0 + 0.8 \\
0.8 \times 2 &= 1 + 0.6 \\
0.6 \times 2 &= 1 + 0.2 \\
0.2 \times 2 &= 0 + 0.4 \\
0.4 \times 2 &= 0 + 0.8 \\
0.8 \times 2 &= 1 + 0.6 \\
0.6 \times 2 &= 1 + 0.2 \\
&\vdots \\
0.0\overline{0011}
\end{aligned}
$$

  This sequences repeats infinite times!

  0.1 is not a machine number, which means, it may not be exactly represented in a computing system.

Floating-point rounding

- Our floating-point representation will have to be as close as possible to 0.1_{10}
  $0.0001100110011001100110011001100110011001\ldots_2$
- Remember that mantissa is 23 and 52 bits for single- and double-precision, respectively.
- IEEE employs rounding.
  - If first extra bit is 1, we add 1 to the rest of the mantissa bits – This is rounding up
  - If first extra bit is 0, we drop all extra bits – This is rounding down.
  - Special case if extra bits are 1000....000
    - Round up if last mantissa bit is 1
    - Round down if last mantissa bit is 0

Floating-point rounding

- Rounded normalized value:

$$1.10011001100110011001101_2 \times 2^{-4}$$

We rounded up for this example

$$1.10011001100110011001101_2 \times 2^{123-127} \text{ (single)}$$

- Floating-point representation:

Single: 1 01111011 10011001100110011001101

- Which represents the value of

0.100000001490116119384765625

- Similarly, double precision represents 0.1 as

0.1000000000000000055511151231257827021181583404541015625

Floating-point rounding

- In your favourite programming language try the following code using float or double data types.

  0.1 + 0.1 + 0.1 == 0.3

  Is the result TRUE or FALSE?

- Problems with accuracy
  - Several failures (in some cases with fatal consequences) have been reported due to numerical errors.

    http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html

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

Concluding Remarks

- ISAs support arithmetic operations
  - Signed and unsigned integers.
  - Floating-point approximation to reals.
- Bounded range and precision
  - Operations can overflow and underflow