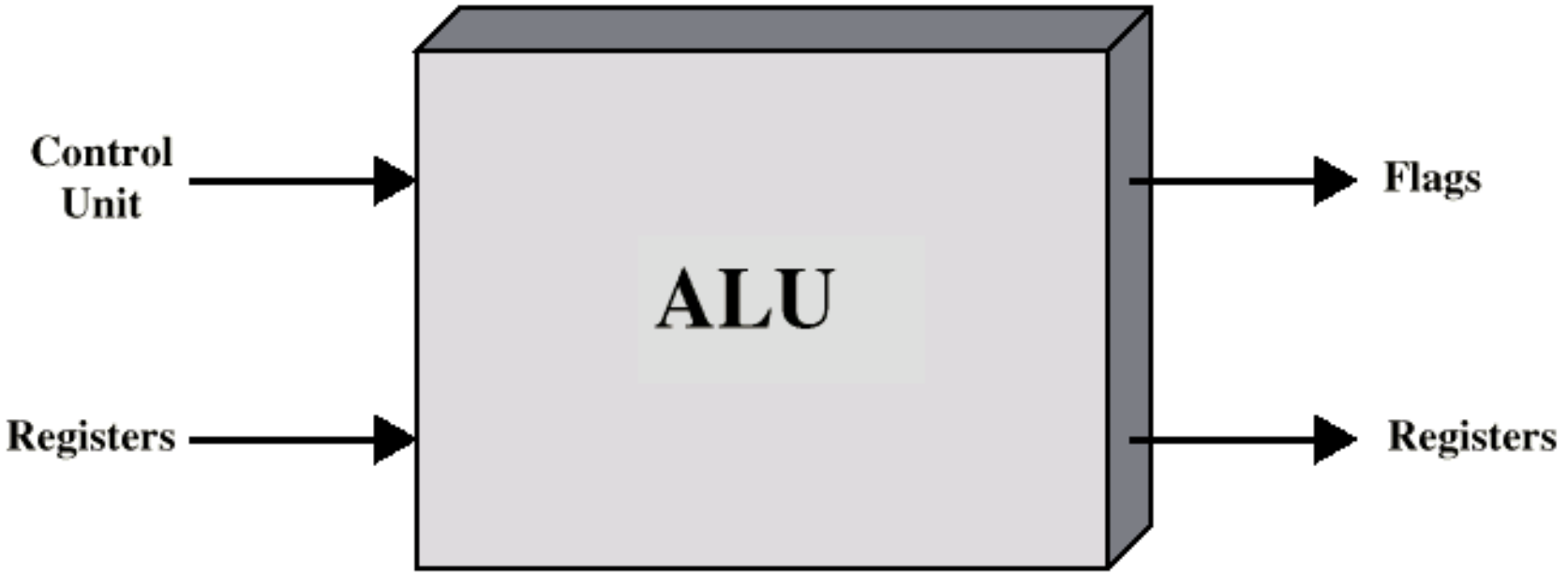
Computer Arithmetic

Arithmetic & Logic Unit

- Does the calculations
- Everything else in the computer is there to service this unit
- Handles integers
- May handle floating point (real) numbers
- May be separate FPU (maths co-processor)
- May be on chip separate FPU (486DX +)

ALU Inputs and Outputs



Conversion Between Lengths

- **Signed Extension**
- Positive number pack with leading zeros
- +18 = 00010010
- +18 = 00000000 00010010
- Negative numbers pack with leading ones
- -18 = 10010010
- -18 = 11111111 10010010
- i.e. pack with MSB (sign bit)

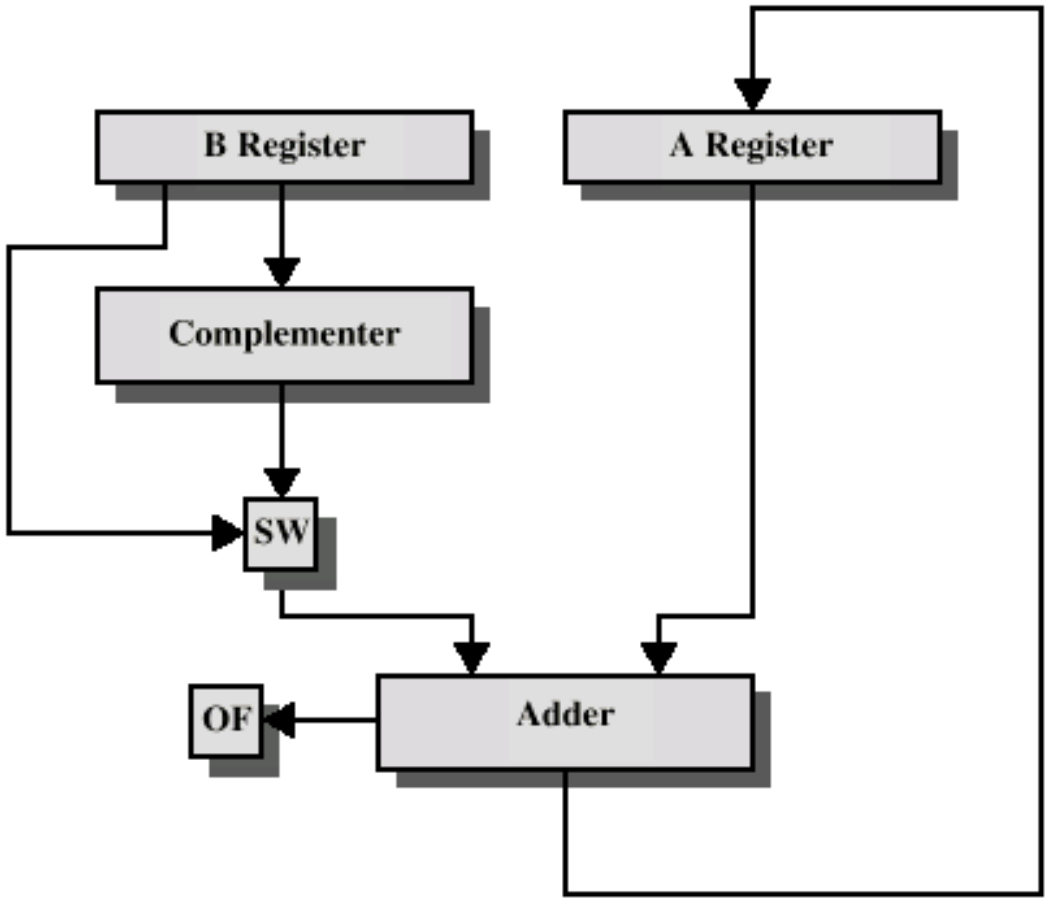
Addition and Subtraction

- Normal binary addition
- Monitor sign bit for overflow

- Take twos compliment of substahend and add to minuend
  - i.e. a - b = a + (-b)

- So we only need addition and complement circuits

Hardware for Addition and Subtraction



OF = overflow bit
SW = Switch (select addition or subtraction)

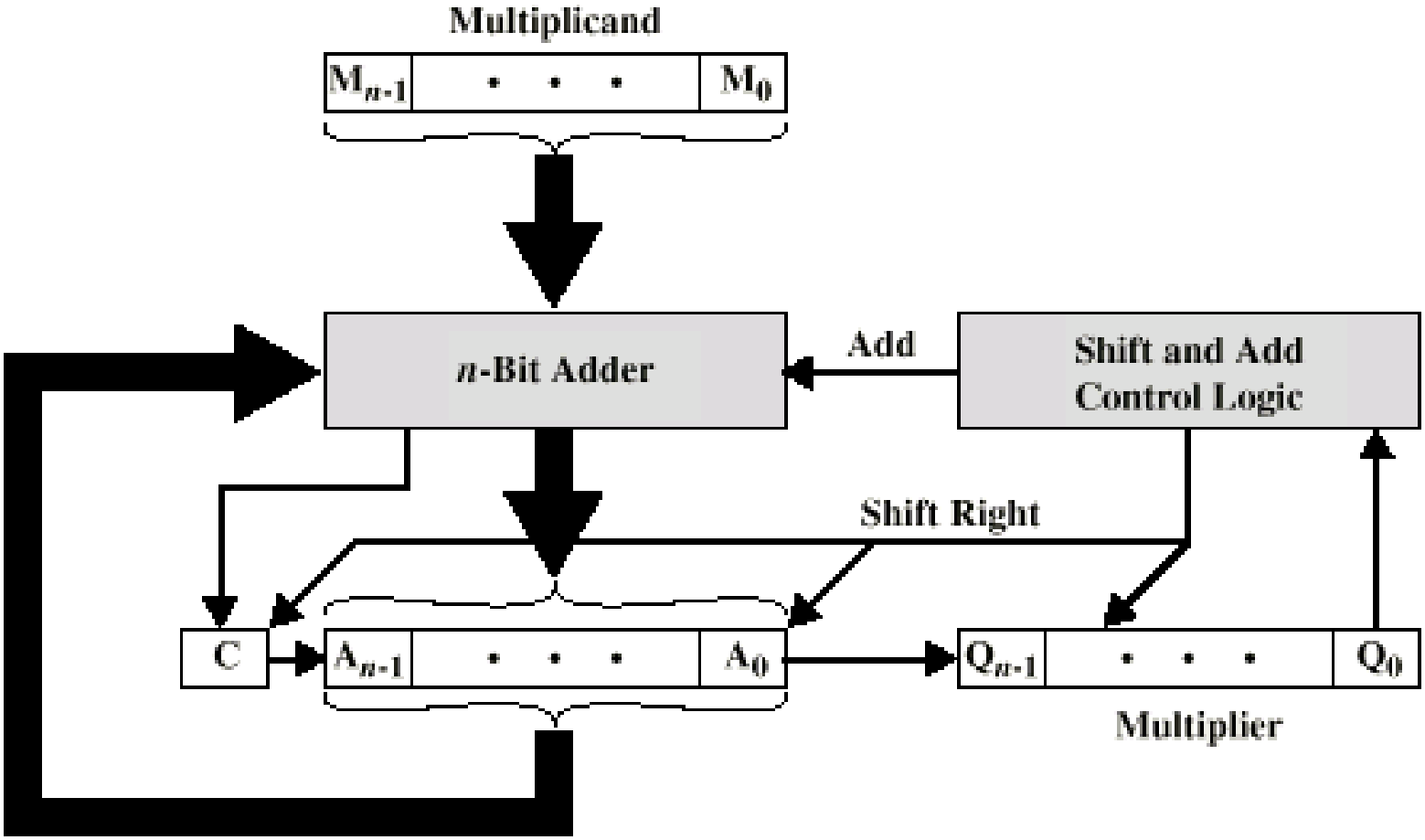
Multiplication

- Complex
- Work out partial product for each digit
- Take care with place value (column)
- Add partial products

Multiplication Example

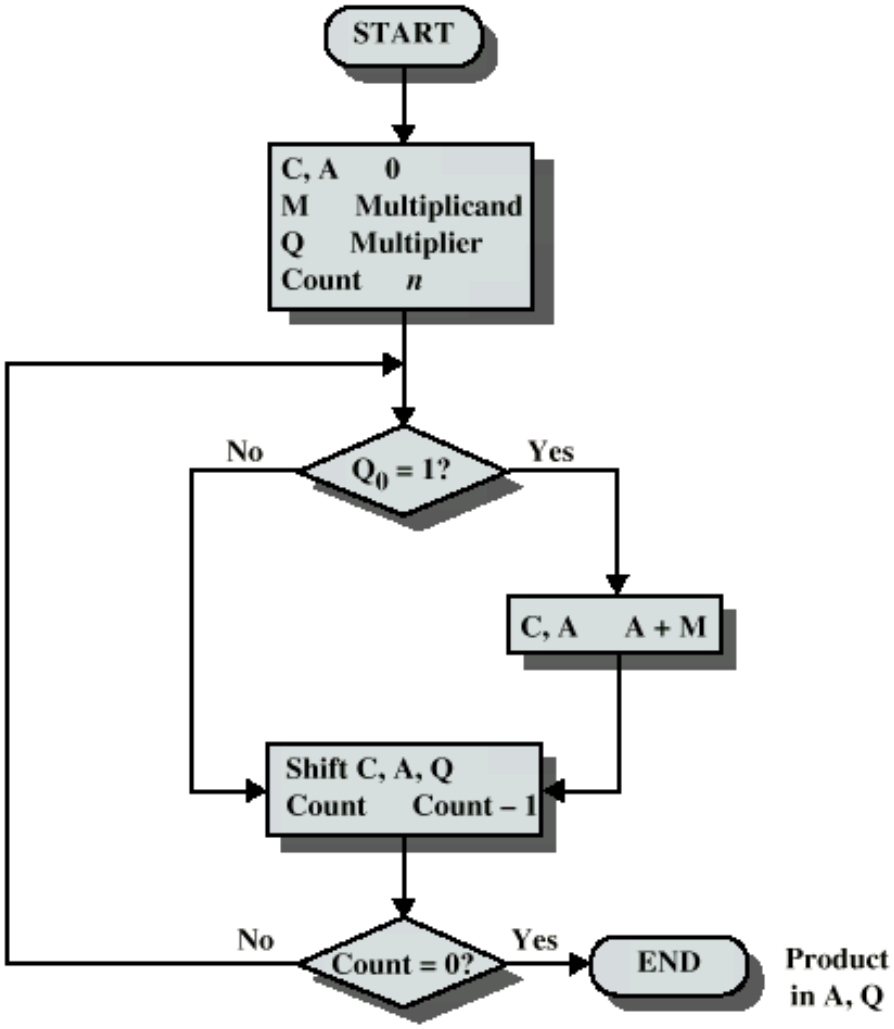
- 1011 Multiplicand (11 dec)
- x 1101 Multiplier (13 dec)
- 1011 Partial products
- 0000 Note: if multiplier bit is 1 copy
- 1011 multiplicand (place value)
- 1011 otherwise zero
- 10001111 Product (143 dec)
- Note: need double length result

Unsigned Binary Multiplication



(a) Block Diagram

Flowchart for Unsigned Binary Multiplication



Simple Right Shift

0 0 0 0 1 1 1 0

↘

0 0 0 0 0 1 1 1

---

1 0 0 0 1 1 1 0

↘

0 1 0 0 0 1 1 1

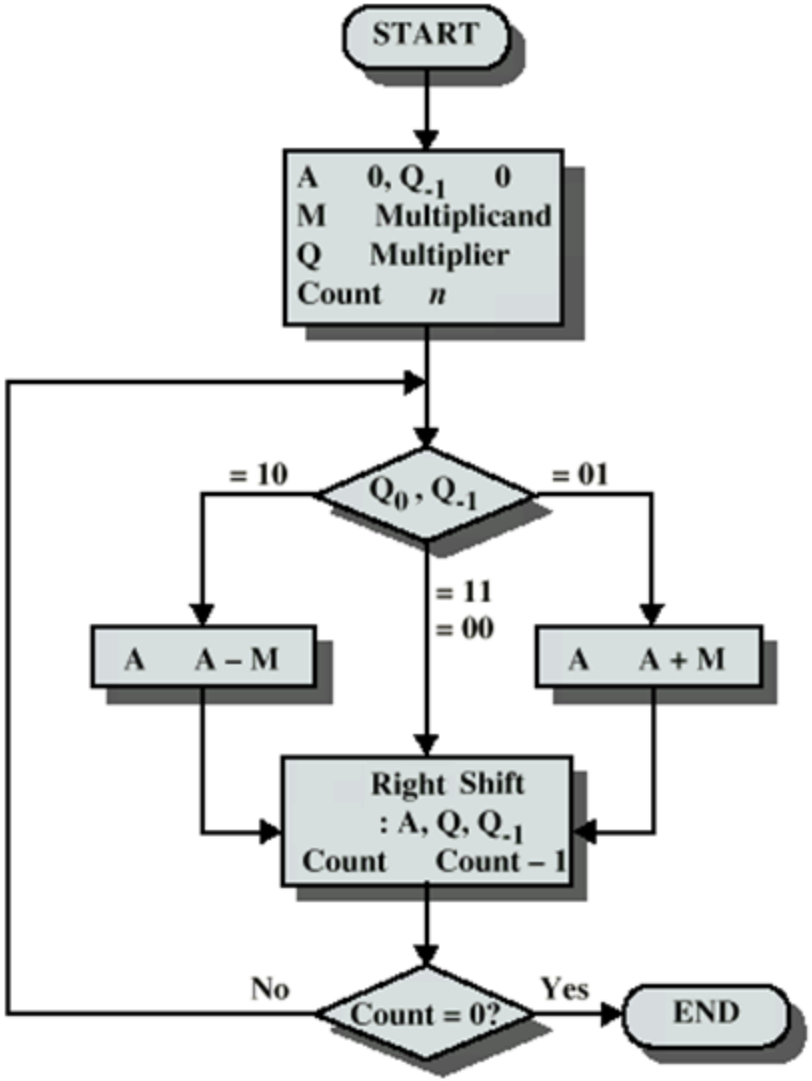
Execution of Example

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

Multiplying Negative Numbers

- This does not work!
- Solution 1
  - —Convert to positive if required
  - —Multiply as above
  - —If signs were different, negate answer
- Solution 2
  - —Booth's algorithm

Booth's Algorithm with simple Right Shift



Example of Booth's Algorithm (-4*3)

Mc		Mp
-4	*	3
M		Q

M	1100
–M	0100

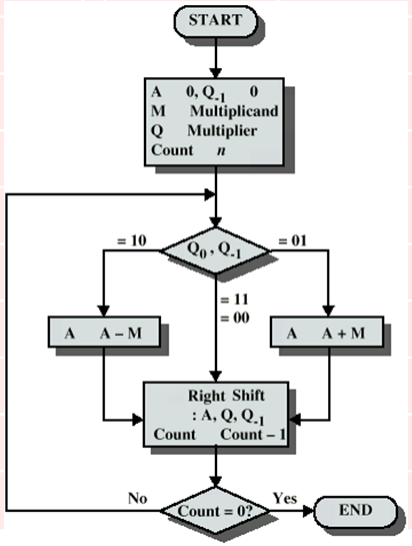
Counter	A				Q				Q_{-1}	Operation
					Q_3	Q_2	Q_1	Q_0		
4	0	0	0	0	0	0	1	1	0	Initialization
	0	1	0	0	0	0	1	1	0	A=A-M
	0	0	1	0	0	0	0	1	1	SRS
3	0	0	1	0	0	0	0	1	1	CD
	0	0	0	1	0	0	0	0	1	SRS
2	0	0	0	1	0	0	0	0	1	CD
	1	1	0	1	0	0	0	0	1	A=A+M
	0	1	1	0	1	0	0	0	0	SRS
1	0	1	1	0	1	0	0	0	0	CD
	0	0	1	1	0	1	0	0	0	SRS
0	0	0	1	1	0	1	0	0	0	CD
										STOP

Mc : Multiplicand (M)

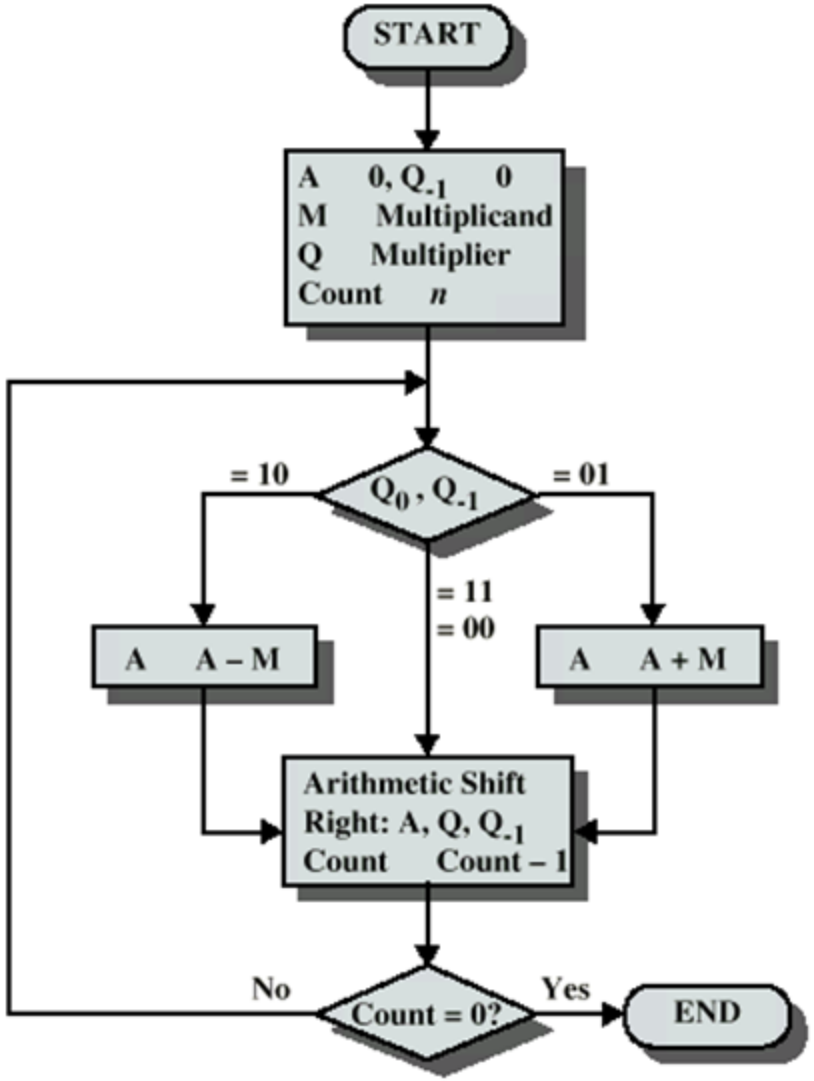
Mp : Multiplier (Q)

SRS : Simple Right Shift

CD : Counter Decrement



Booth's Algorithm using Arithmetic Right Shift



Use Simple Right Shift
OR
Arithmetic Right Shift

---

0 0 0 0 1 1 1 0

↘

0 0 0 0 0 1 1 1

---

1 0 0 0 1 1 1 0

↘

1 1 0 0 0 1 1 1

Reason: Sign extension in 2's Complement

Note: You can get Answer using both shift..
But simple right shift will need some manipulation in answer in the end

Example of Booth's Algorithm using ARS (7*3)

Mc Mp

7 * 3

M Q

M 0111

–M 1001

Counter	A				Q				Q_{-1}	Operation
					Q_3	Q_2	Q_1	Q_0		
4	0	0	0	0	0	0	1	1	0	Initialization
	1	0	0	1	0	0	1	1	0	A= A-M
	1	1	0	0	1	0	0	1	1	ARS
3	1	1	0	0	1	0	0	1	1	CD
	1	1	1	0	0	1	0	0	1	ARS
2	1	1	1	0	0	1	0	0	1	CD
	0	1	0	1	0	1	0	0	1	A= A+M
	0	0	1	0	1	0	1	0	0	ARS
1	0	0	1	0	1	0	1	0	0	CD
	0	0	0	1	0	1	0	1	0	ARS
0	0	0	0	1	0	1	0	1	0	CD
										STOP

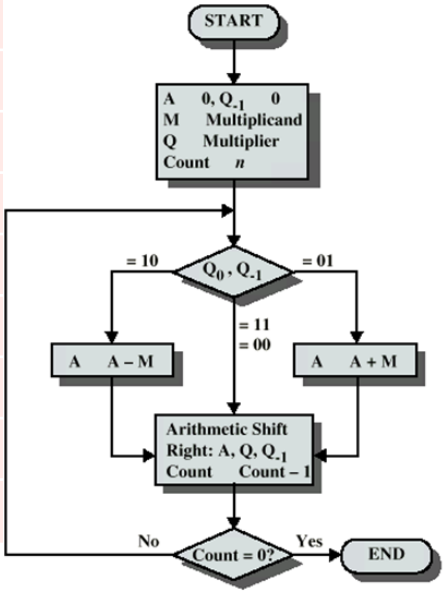
Mc : Multiplicand (M)

Mp : Multiplier (Q)

ARS : Arithmatic Right Shift

CD : Counter Decrement

AQ : Answer



Example of Booth's Algorithm using ARS (-7*3)

Mc		Mp
-7	*	3
M		Q

M 1001

–M 0111

Counter	A				Q				Q_{-1}	Operation
					Q_3	Q_2	Q_1	Q_0		
4	0	0	0	0	0	0	1	1	0	Initialization
	0	1	1	1	0	0	1	1	0	A=A-M
	0	0	1	1	1	0	0	1	1	ARS
3	0	0	1	1	1	0	0	1	1	CD
	0	0	0	1	1	1	0	0	1	ARS
2	0	0	0	1	1	1	0	0	1	CD
	1	0	1	0	1	1	0	0	1	A=A+M
	1	1	0	1	0	1	1	0	0	ARS
1	1	1	0	1	0	1	1	0	0	CD
	1	1	1	0	1	0	1	1	0	ARS
0	1	1	1	0	1	0	1	1	0	CD
										STOP

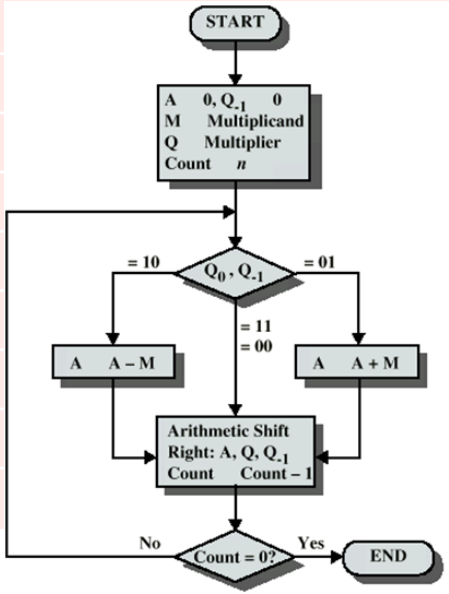
Mc : Multiplicand (M)

Mp : Multiplier (Q)

ARS : Arithmatic Right Shift

CD : Counter Decrement

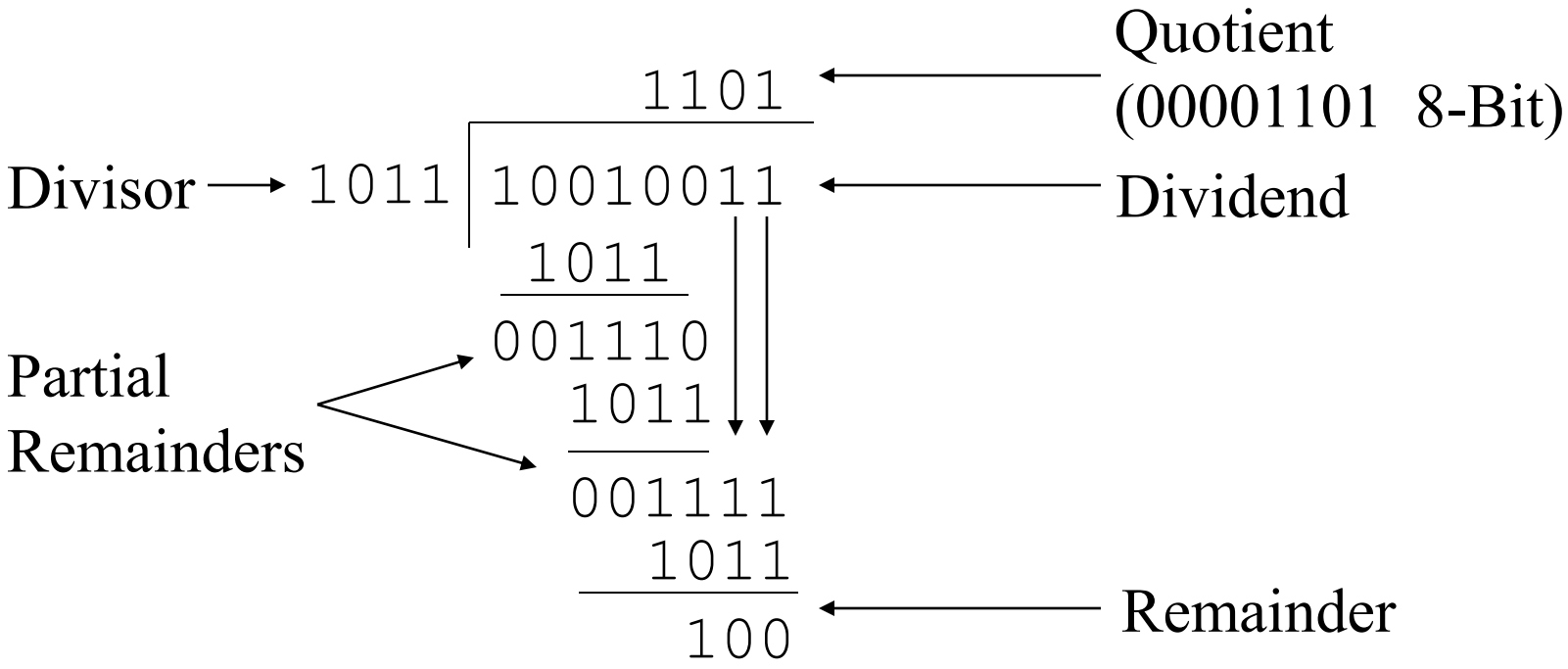
AQ : Answer



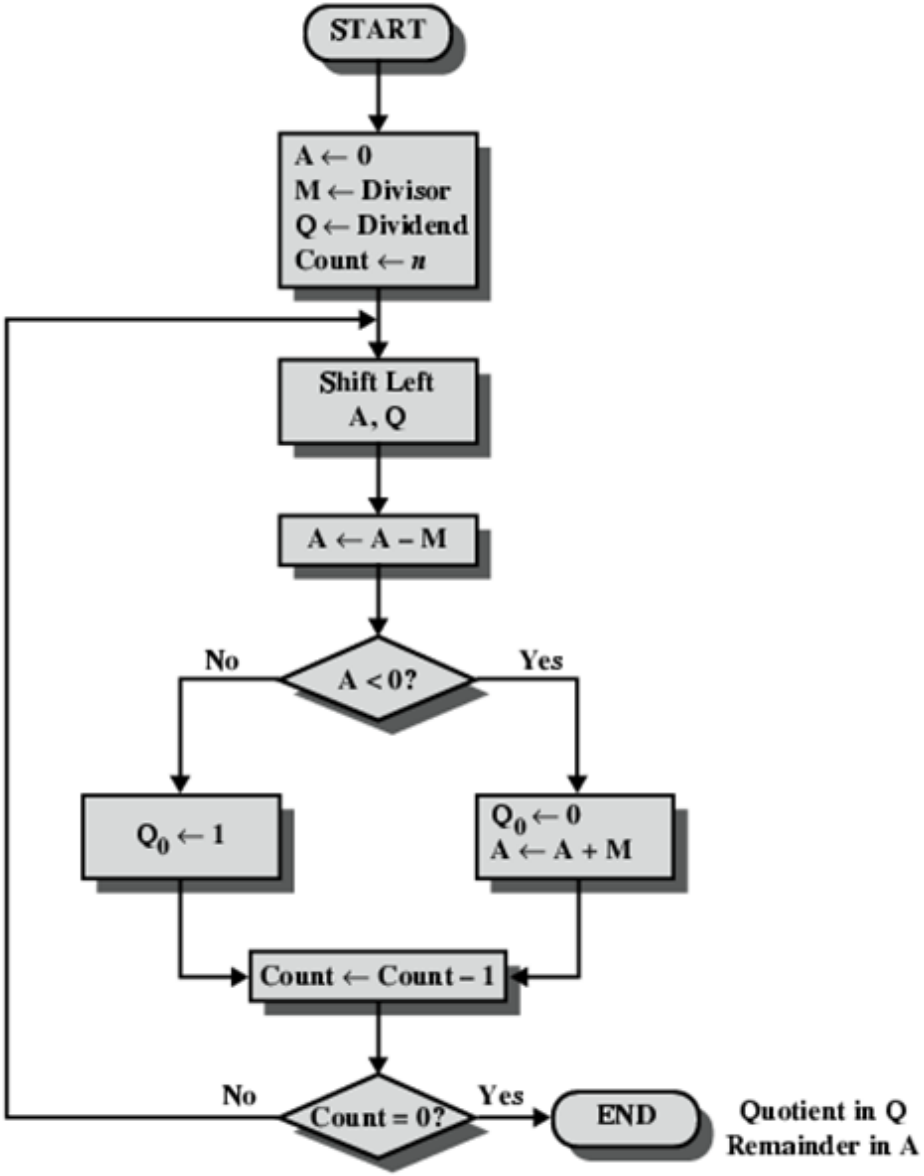
Division

- More complex than multiplication
- Negative numbers are really bad!
- Based on long division

Division of Unsigned Binary Integers



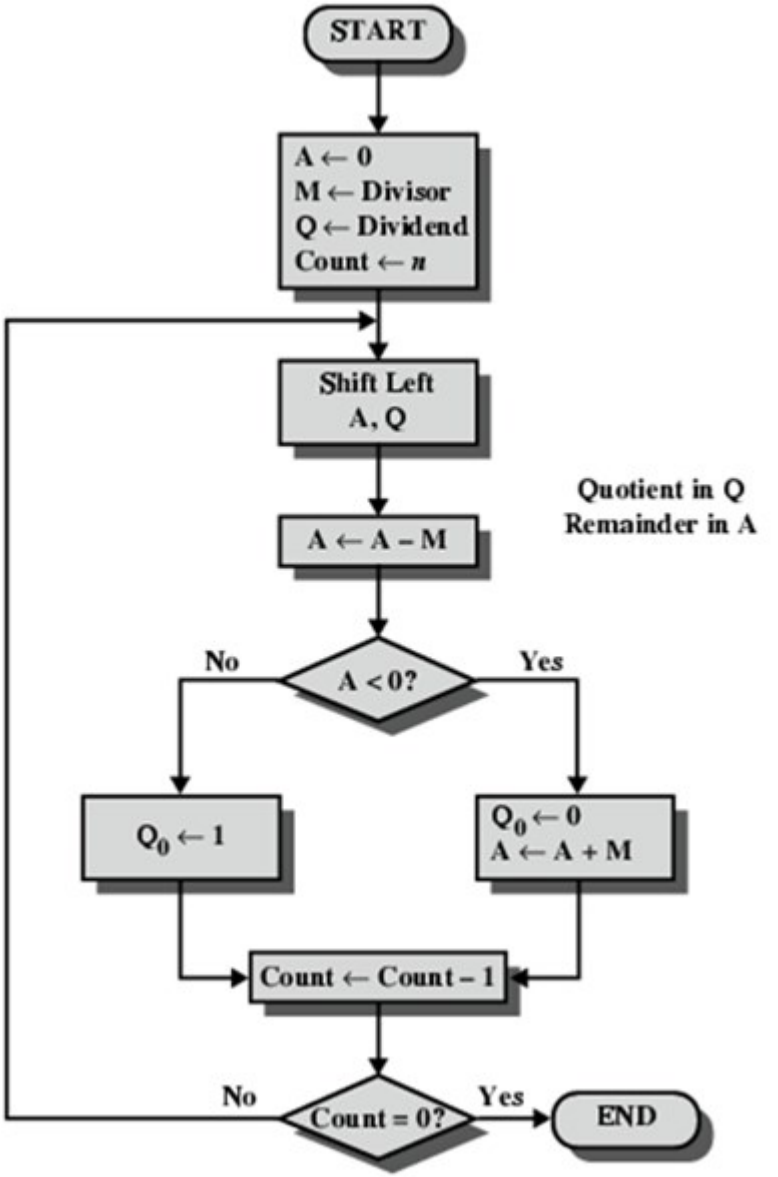
Flowchart for Unsigned Binary Division



Binary Division: Example

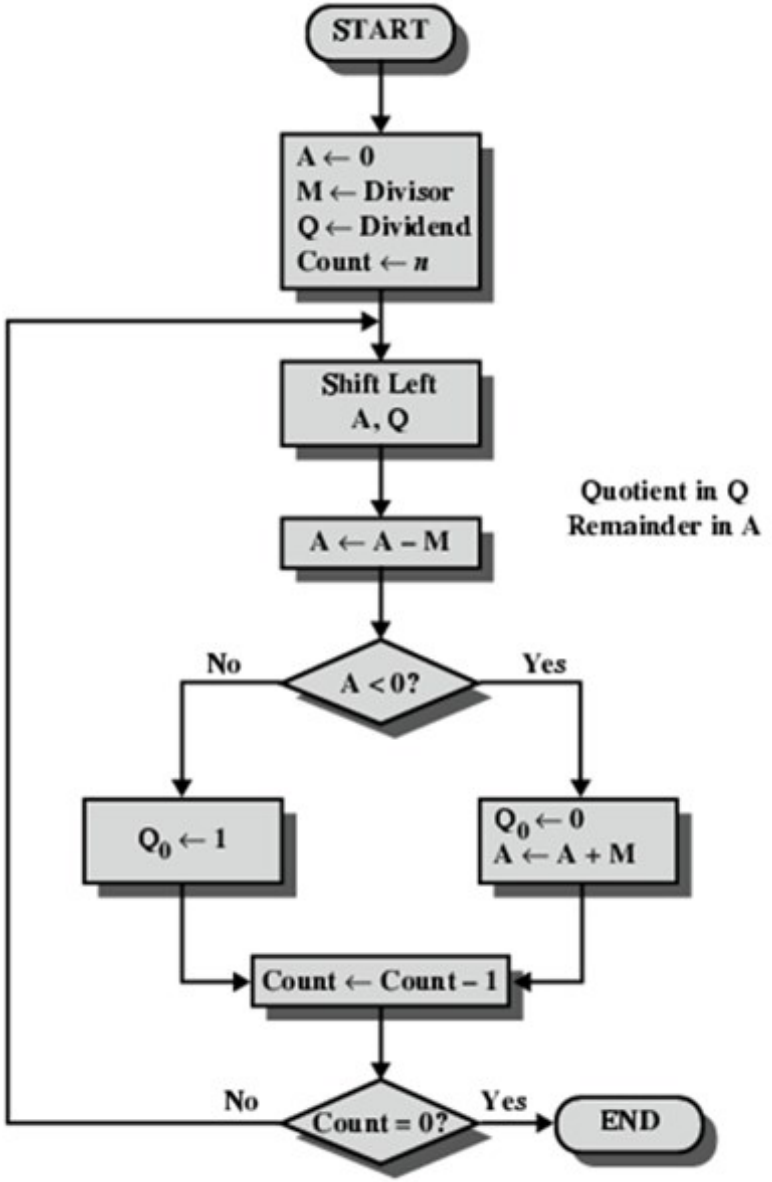
Dividend (Q) = 11, Devisor(M) = 3

n	M	A	Q	Operation
4	0011	0000	1011	initialize
	0011	0001	011_	shift left AQ
	0011	1110	011_	A=A-M (Using 2's Compl)
	0011	0001	0110	Q[0]=0 and
				Restore A i.e. A=A+M
3	0011	0010	110_	shift left AQ
	0011	1111	110_	A=A-M
	0011	0010	1100	Q[0]=0, Restore A
2	0011	0101	100_	shift left AQ
	0011	0010	100_	A=A-M
	0011	0010	1001	Q[0]=1
1	0011	0101	001_	shift left AQ
	0011	0010	001_	A=A-M
	0011	0010	0011	Q[0]=1



Problem

1. 2 / 2
2. 4 / 2



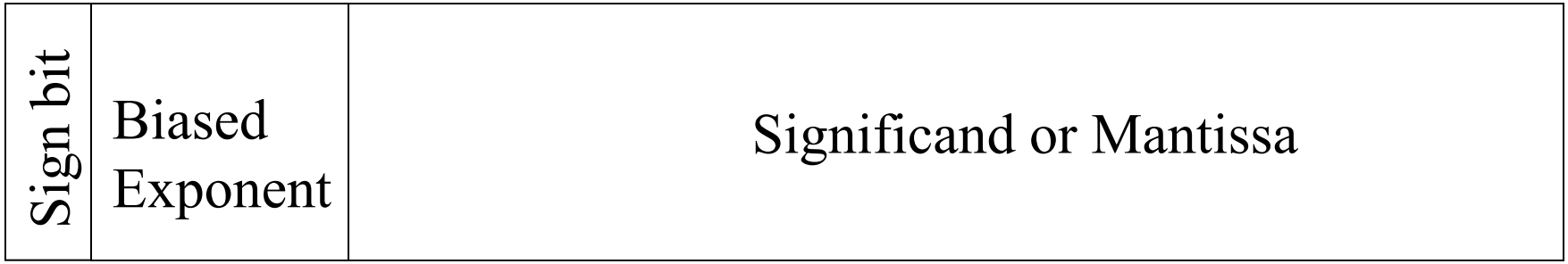
Division of signed numbers

- Use previously taught division algorithm and adjust sign later

Real Numbers

- Numbers with fractions
- Could be done in pure binary
  - —$1001.1010 = 2^4 + 2^0 + 2^{-1} + 2^{-3} = 9.625$
- Where is the binary point?
- Fixed?
  - —Very limited
- Moving?
  - —How do you show where it is?

Floating Point



- +/- .significand x $2^{exponent}$
- Misnomer
- Point is actually fixed between sign bit and body of mantissa
- Exponent indicates place value (point position)

Floating Point Examples

(a) Format

(b) Examples

$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.638125 \times 2^{20}$

$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.638125 \times 2^{20}$

$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.638125 \times 2^{-20}$

$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.638125 \times 2^{-20}$

Signs for Floating Point

- Mantissa is stored in 2s compliment
- Exponent is in excess or biased notation
  - e.g. Excess (bias) 128 means
  - 8 bit exponent field
  - Pure value range 0-255
  - Subtract 128 to get correct value
  - Range -128 to +127

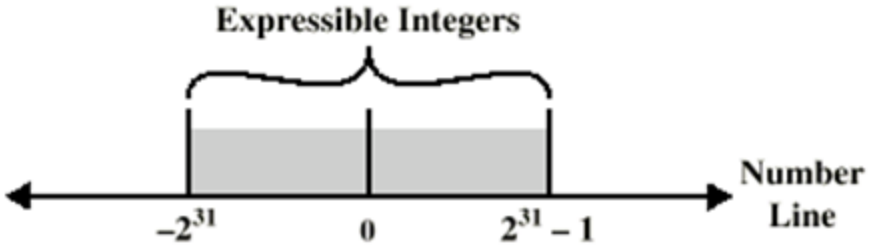
Normalization

- FP numbers are usually normalized
- i.e. exponent is adjusted so that leading bit (MSB) of mantissa is 1
- Since it is always 1 there is no need to store it
- (c.f. Scientific notation where numbers are normalized to give a single digit before the decimal point
- e.g. 3.123×10^3)

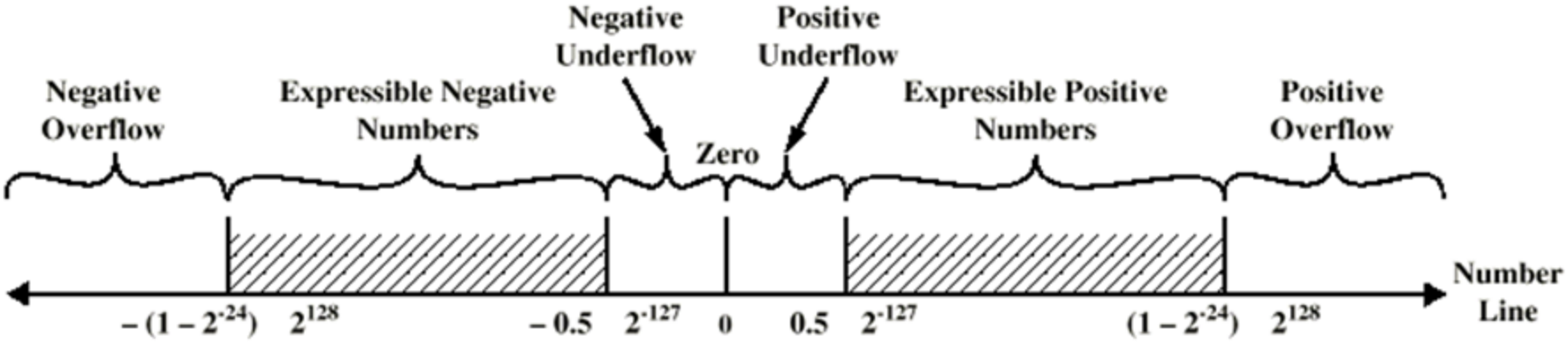
FP Ranges

- For a 32 bit number
  - —8 bit exponent
  - —+/- $2^{256} \approx 1.5 \times 10^{77}$
- Accuracy
  - —The effect of changing lsb of mantissa
  - —23 bit mantissa $2^{-23} \approx 1.2 \times 10^{-7}$
  - —About 6 decimal places

Expressible Numbers

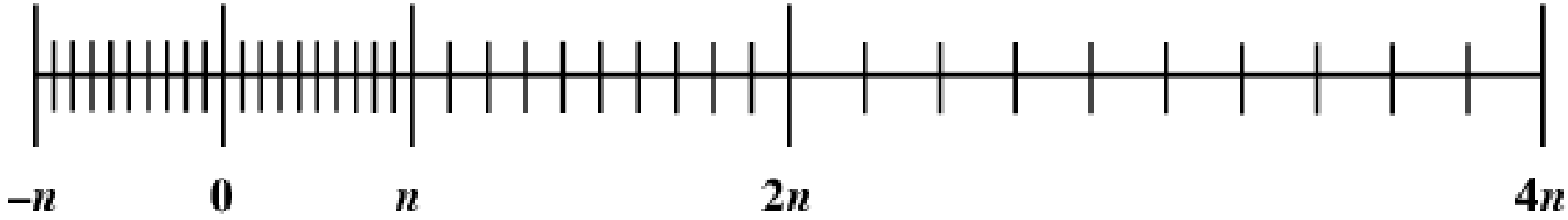


(a) Twos Complement Integers



(b) Floating-Point Numbers

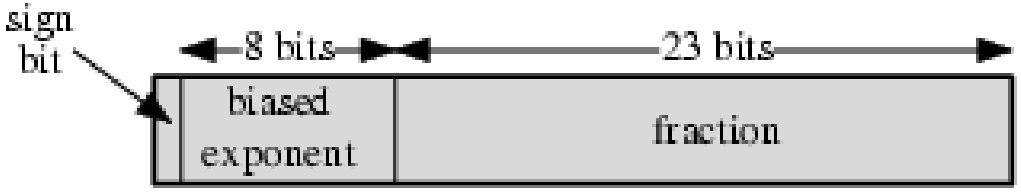
Density of Floating Point Numbers



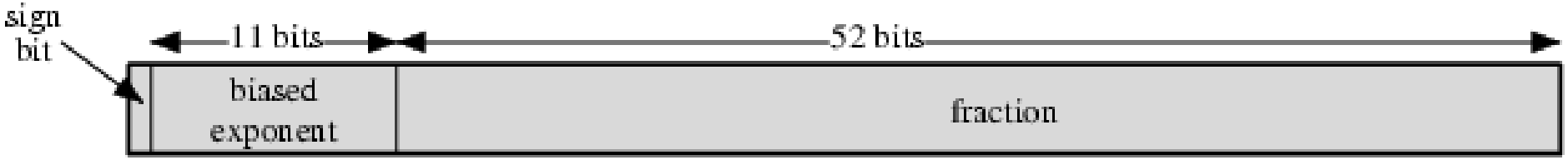
IEEE 754

- Standard for floating point storage
- 32 and 64 bit standards
- 8 and 11 bit exponent respectively
- Extended formats (both mantissa and exponent) for intermediate results

IEEE 754 Formats



(a) Single format

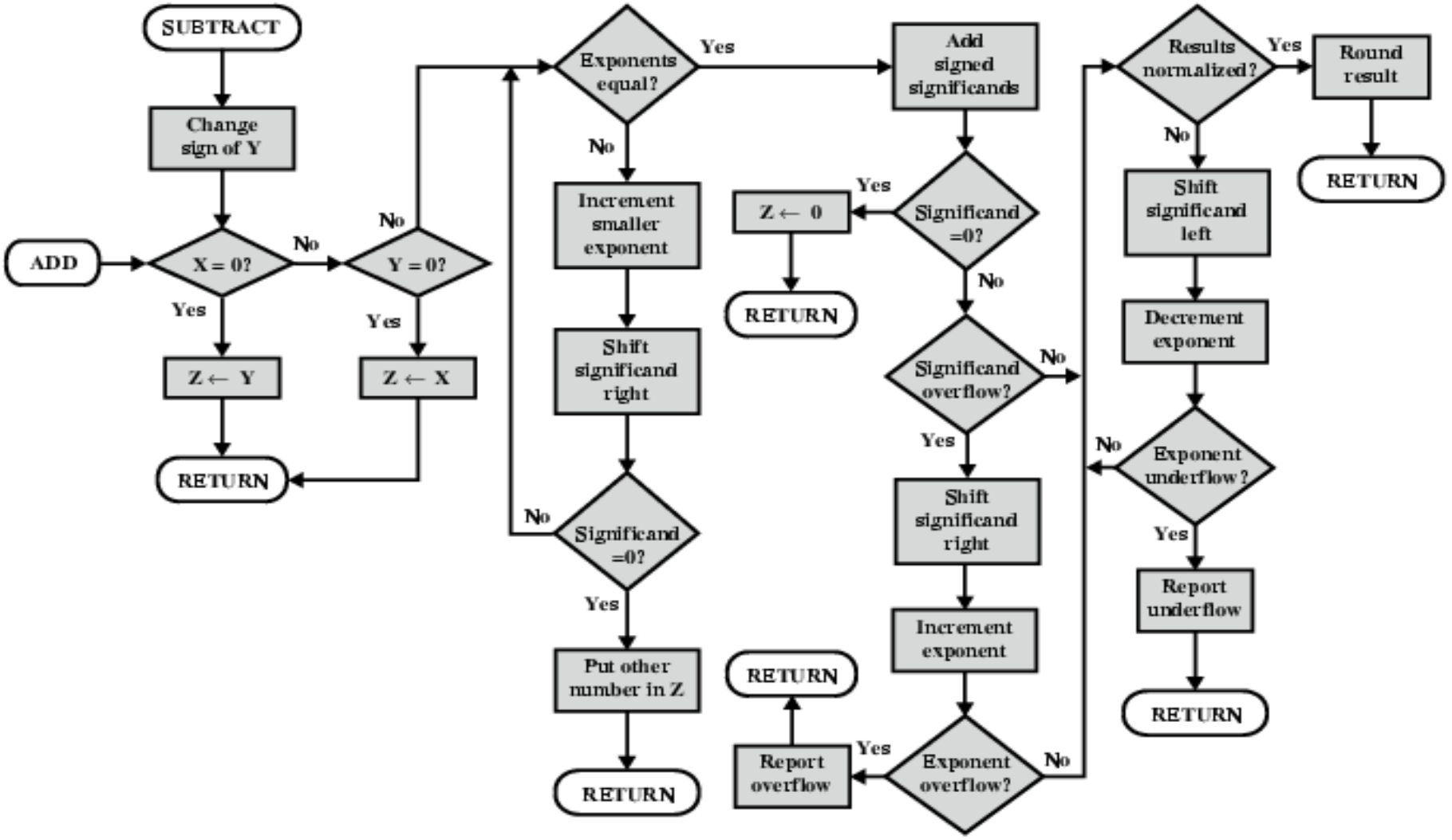


(b) Double format

FP Arithmetic +/-

- Check for zeros
- Align significands (adjusting exponents)
- Add or subtract significands
- Normalize result

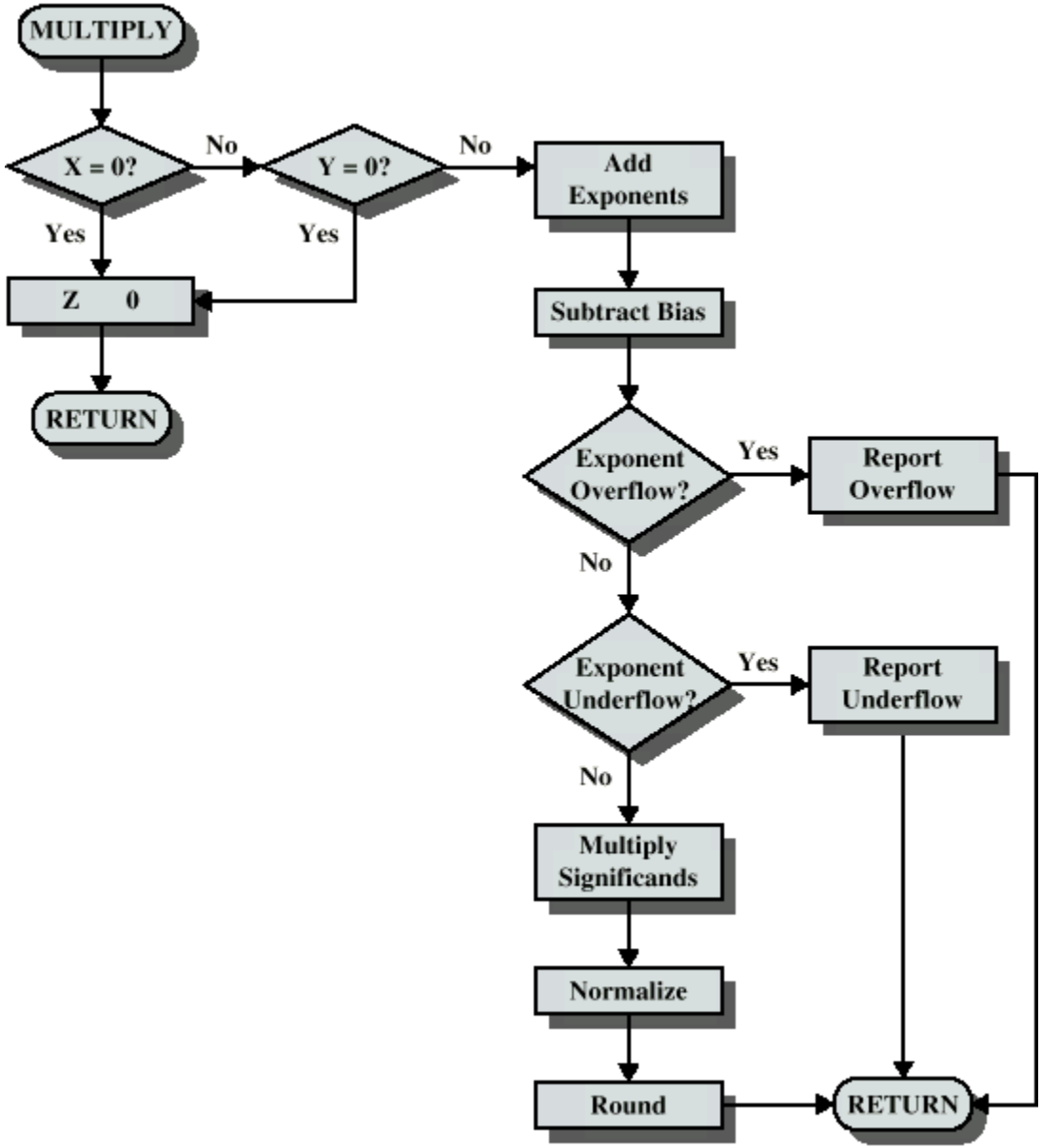
FP Addition & Subtraction Flowchart



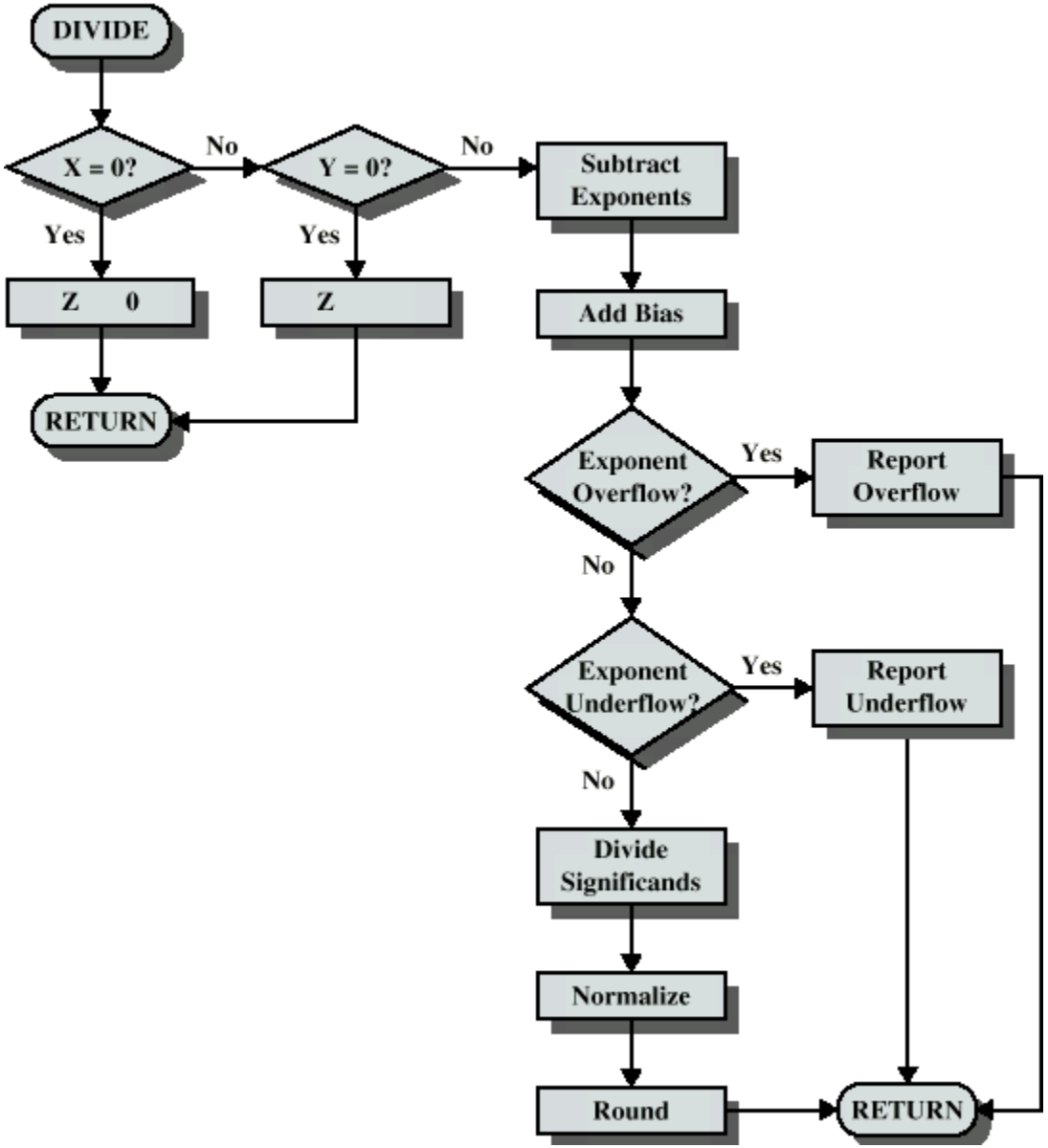
FP Arithmetic x/÷

- Check for zero
- Add/subtract exponents
- Multiply/divide significands (watch sign)
- Normalize
- Round
- All intermediate results should be in double length storage

Floating Point Multiplication



Floating Point Division



Required Reading

- William Stallings Chapter 9
- IEEE 754 on IEEE Web site