Division

1. Align dividend and divisor with their most significant digits
2. test how many times n the divisor fits into the locally aligned dividend
3. n is the value of the quotient digit
4. subtract divisor n times from the locally aligned dividend
5. extend local remainder by the next less significant digit of the dividend, thus forming a new local dividend
6. repeat steps 2.-5. until all digits of the dividend are considered
7. the local remainder after the last subtraction is the remainder of the division

Example (decimal):  \[ 195 \div 13 = 15 \]

\[
\begin{array}{c|c|c|c}
\text{Divisor} & \text{1} & \text{9} & \text{5} \\
\hline
\text{Dividend} & \text{1} & \text{3} & \text{\textasciitilde Quotient} \\
\hline
& \text{6} & \text{5} & \text{\textasciitilde Dividend} \\
\hline
& \text{6} & \text{5} & \text{\textasciitilde Remainder} \\
\hline
\end{array}
\]

Example (binary):

\[ 19 \div 6 = 3 \text{ r e m } 1 \]

\[
\begin{array}{c|c|c|c}
\text{3} & \text{0} & \text{1} & \text{1} \\
\hline
\text{6} & \text{1} & \text{1} & \text{0} \\
\hline
\text{1} & \text{0} & \text{0} & \text{1} \\
\hline
\text{0} & \text{1} & \text{1} & \text{1} \\
\hline
\text{0} & \text{0} & \text{1} & \text{1} \\
\hline
\end{array}
\]

Sequential Restoring Division

- A shift register keeps both the (remaining) dividend as well as the quotient
- with each cycle, dividend decreases by one digit & quotient increases by one digit
- the MSB’s of the remaining dividend and the divisor are aligned in each cycle
- major difference to multiplication:
  1. we do not know if we can subtract the divisor or not
  2. if the subtraction failed, we have to restore the original dividend

Sequential Restoring Division

1. Load the 2's dividend into both halves of shift register, and add a sign bit to the left
2. add a sign bit to the left of the divisor
3. generate the 2's complement of the divisor
4. shift to the left
5. add 2's complement of the divisor to the upper half of the shift register including sign bit (subtract)
6. if sign of the result is cleared (positive)
   - then set LSB of the lower half of the shift register to one
   - else clear LSB of the lower half and add the divisor to upper half of shift register (restore)
7. repeat from 4. and perform the loop n times
8. after termination:
   - lower half of shift register \( \Rightarrow \) quotient
   - upper half of shift register \( \Rightarrow \) remainder
Division Overflow

\[ 78 \div 3 = 26 \]

Non-Restoring Division

- Restoring: Each time subtraction fails,
  1. the divisor is added
  2. the dividend is shifted left
  3. the divisor is subtracted
- the result equals \( 2(d + v) - v \)
- \( v \) can be added after shifting, thus replacing the subtraction in the next cycle
- this equals \( 2d + v \)
- both variants deliver identical results, but non-restoring saves one subtraction step!

⇒ An overflow occurs if any \( n \)-bit remainder is greater or equal to the \( n \)-bit divisor

Combinational Network

“Compare and Subtract if Greater”

Conclusions for Division

- A sequential division can be implemented with the same hardware as used for sequential multiplication
- however, less potential for optimization
- non-restoring division better performance
- division overflow:
  1. an \( n \)-bit \( \times n \)-bit multiplication always fits into a \( 2n \)-bit result
  2. a division of \( 2n \)-bit by \( n \)-bit may not be representable in \( n \) bits
Summary Integer Arithmetic

1. Number representation:
   - Signed vs. unsigned
   - Sign and magnitude
   - 1’s complement
   - 2’s complement

2. Addition:
   - Carry propagation ⇒ performance?
   - Ripple-Carry adder (intuitive)
   - Carry-Lookahead adder (efficient)

3. Subtraction:
   - Special case of addition

4. Multiplication:
   - Ripple-Carry per row (intuitive)
   - Carry-Save adder tree (efficient)
   - Booth recoding
   - Bit pairing (very efficient)
   - Sequential methods (cheap but slow)

5. Division:
   - Overflow may occur
   - No Booth recoding / bit pairing
   - Restoring sequential division
   - Non-restoring (more efficient)
   - Full network ⇒ rippling (no gain)

Integers and Fractions

- Integer representations are sufficient only for simple tasks
- Scientific calculations require much larger range of numbers and fractions

1. Split number into integer and fraction parts of pre-defined widths (fixed-point)
2. Define number of significant digits along with an indicator – the exponent – where to set the point (floating-point)

Fixed-Point Representation

⇒ Constant positions of integer and fraction
⇒ Single integer with constant multiplicator

- Example 1: Decimal value \( v \) containing 6 digits after the point, possible representation of \( r \):
  \[ r = v \times 10^6 \]

- Example 2: 32-bit 2's-complement representation, integer and fraction of 16-bit width each:
  \[ r = v \times 2^{16} \]

Floating-Point Representation

Scale factor no longer a constant but a function of variable \( e \)

\[ r = v \times 10^e \]

where \( v \) is the mantissa and \( e \) is the exponent

<table>
<thead>
<tr>
<th>Value</th>
<th>Fixed</th>
<th>Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>+1.500000</td>
<td>+1.500 e+00</td>
</tr>
<tr>
<td>-1000</td>
<td>-1000.000000</td>
<td>-1.000 e+03</td>
</tr>
<tr>
<td>( \frac{1}{10} )</td>
<td>+0.033333</td>
<td>+3.333 e-02</td>
</tr>
<tr>
<td>( -10^{-48} )</td>
<td>+0.000000</td>
<td>-1.000 e-48</td>
</tr>
</tbody>
</table>

\( r \) is normalized if the decimal point is placed immediately after first significant digit

Binary Floating-Point

...similar to the decimal one

\[ r = v \times 2^e \]

<table>
<thead>
<tr>
<th>Value (base-10)</th>
<th>Float (base-2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>1.5 \times 2^{10}</td>
</tr>
<tr>
<td>-1000</td>
<td>-1.95313 \times 2^{19}</td>
</tr>
<tr>
<td>( \frac{1}{10} )</td>
<td>1.06667 \times 2^{-5}</td>
</tr>
</tbody>
</table>

Note: In the normalized binary representation, there is always a 1 in the first significant digit!