**Binary division**

- quotient = dividend / divisor, with a remainder
- dividend = divisor × quotient + remainder
- Given dividend and divisor, we want to obtain quotient (Q) and remainder (R)
- We will start from our paper & pencil method

---

**Hardware design 1**

[Diagram of hardware design 1 with 64-bit shift register, 64-bit ALU, 32-bit shift register, and control flow]
Hardware design 2

- 32-bit ALU
- Divisor
- 64-bit shift register
- Quotient
- Shift left

Hardware design 3

1. Place dividend here first
2. Run the algorithm
3. Find remainder here
4. Find quotient here
## Example

- Let's do 0111/0010 (7/2) – unsigned

### Hardware design 3

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Step</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>initial values</td>
<td>0000 0111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift remainder left by 1</td>
<td>0000 1110</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>remainder = remainder - divisor</td>
<td>1110 1110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&lt;0) =&gt; +divisor; shift left; r0=0</td>
<td>0001 1100</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>remainder = remainder - divisor</td>
<td>1111 1100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&lt;0) =&gt; +divisor; shift left; r0=0</td>
<td>0011 1000</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>remainder = remainder - divisor</td>
<td>0001 1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&gt;0) =&gt; shift left; r0=1</td>
<td>0011 0001</td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>remainder = remainder - divisor</td>
<td>0001 0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&gt;0) =&gt; shift left; r0=1</td>
<td>0010 0011</td>
</tr>
<tr>
<td>done</td>
<td>0010</td>
<td>shift “left half of remainder” right by 1</td>
<td>0001 0011</td>
</tr>
</tbody>
</table>

## Exercise sheet

### Hardware design 3

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Step</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>initial values</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift remainder left by 1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>0010</td>
<td>shift “left half of remainder” right by 1</td>
<td></td>
</tr>
</tbody>
</table>
Restoring division

- The three hardware designs we saw are based on the notion of “restoring division”
  - At first, attempt to subtract divisor from dividend
  - If the result of subtraction is negative – it rolls back by adding divisor
    - This step is called “restoring”

- It’s a “trial-and-error” approach; can we do better?

Non-restoring division

- Let’s revisit the restoring division designs
  - Given remainder $R$ ($R<0$) after subtraction
  - By adding divisor $D$ back, we have $(R+D)$
  - After shifting the result, we have $2 \times (R+D) = 2 \times R + 2 \times D$
  - If we subtract the divisor in the next step, we have $2 \times R + 2 \times D - D = 2 \times R + D$

- This is equivalent to
  - Left-shifting $R$ by 1 bit and then adding $D$!
Example, non-restoring division

- Let's again do 0111/0010 (7/2) – unsigned

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Hardware design 3, non-restoring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Step</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>initial values</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift remainder left by 1</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>remainder = remainder - divisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&lt;0) =&gt; shift left; r0=0</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>remainder = remainder + divisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&lt;0) =&gt; shift left; r0=0</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>remainder = remainder - divisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&gt;0) =&gt; shift left; r0=1</td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>remainder = remainder - divisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(remainder&gt;0) =&gt; shift left; r0=1</td>
</tr>
<tr>
<td>done</td>
<td>0010</td>
<td>shift &quot;left half of remainder&quot; right by 1</td>
</tr>
</tbody>
</table>

Exercise sheet

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Hardware design 3, non-restoring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Step</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>initial values</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift remainder left by 1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>done</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Floating-point (FP) numbers

- Computers need to deal with real numbers
  - Fractional numbers (e.g., 3.1416)
  - Very small numbers (e.g., 0.000001)
  - Very larger numbers (e.g., 2.7596 × 10^9)

- Components in a binary FP number
  - \((-1)^{\text{sign}} \times \text{significand (a.k.a. mantissa)} \times 2^{\text{exponent}}\)
  - More bits in significand gives higher accuracy
  - More bits in exponent gives wider range (exponent +/-)

- A case for FP representation standard
  - Portability issues
  - Improved implementations
  ⇒ IEEE-754

Format choice issues

- Example floating-point numbers (base-10)
  - \(1.4 \times 10^{-2}\)
  - \(-20.0 = -2.00 \times 10^1\)

- What components do we have?
  - \((-1)^{\text{sign}} \times \text{significand (a.k.a. mantissa)} \times 2^{\text{exponent}}\)
    - Sign
    - Significand
    - Exponent (+/-)

- Representing sign is easy (0=positive, 1=negative)
- Significand is unsigned (sign-magnitude)
- Exponent is a signed integer. What method do we use?
IEEE 754

- A standard for representing FP numbers in computers
  - Single precision (32 bits): 8-bit exponent, 23-bit significand
  - Double precision (64 bits): 11-bit exponent, 52-bit significand

- Leading “1” in significand is implicit (why?)
- Exponent is a signed number in biased format
  - “Biased” format – for easier sorting of FP numbers
  - All 0’s is the smallest, all 1’s is the largest
  - Bias of 127 for SP and 1023 for DP
- Hence, to obtain the actual value of a representation
  - \((-1)^{\text{sign}} \times (1.\#\text{significand}) \times 2^{\text{exponent}}\); here “#” is concatenation
  - exponent is a biased number (see next slide)

Biased representation

- Yet another binary number representation
  - Signed number allowed

- 000…000 is the smallest number, 111 ... 111 is largest number!
- To get the real value, subtract a pre-determined “bias” from the unsigned evaluation of the bit pattern
- In other words, \(\text{representation} = \text{value} + \text{bias}\)

- Bias for the “exponent” field in IEEE 754
  - 127 (SP), 1023 (DP)

- E.g., suppose exponent field = 01111101b = 125d
  - b/c we added the bias, we must subtract it to get decimal value
  - thus, exponent in decimal is really: 125d - 127d = -2d
  - what’s the decimal value for 10000111b=135d? (135d - 127d = 8d)
IEEE 754 example

-0.75\text{ten}
- Same as -3/4 or -3/2^2
- In binary, -11\text{two}/2^{\text{ten}} or -0.11\text{two}
- In a normalized form, it’s -1.1\text{two} \times 2^{-1}

- In IEEE 754
- Sign bit is 1 – number is negative!
- Significand is 0.1 – the leading 1 is implicit!
- Exponent is -1; (1 + 127 = 126 in biased representation)
  - 126 is in exponent field, so “decimal exponent value is 126 - 127 = -1

IEEE 754 summary

\begin{tabular}{|c|c|c|c|c|}
\hline
Exponent & Fraction & Exponent & Fraction & Represented Object \\
\hline
0 & 0 & 0 & 0 & 0 \\
0 & non-zero & 0 & non-zero & 0 \\
1-254 & anything & 1-2046 & anything & +/- floating-point numbers \\
255 & 0 & 2047 & 0 & +/- infinity \\
255 & non-zero & 2047 & non-zero & NaN (Not a Number) \\
\hline
\end{tabular}